UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Diseño e implementación de un hub de control domótico

Autor: Pallarés Jiménez, Ignacio
Tutor: Delgado Mohatar, Óscar
Ponente: Anguiano Rey, Eloy

Junio 2019

Pallarés Jiménez, Ignacio
Diseño e implementación de un hub de control domótico

Pallarés Jiménez, Ignacio

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A mi Abuba

“Trata a un ser humano como es, y seguirá siendo lo que es.
Pero trátalo como puede llegar a ser, y se convertirá en lo que está llamado a ser.”

J. W. Goethe

Agradecimientos

A mis padres y hermanos, que creyeron en mí desde el primer momento y me han alentado durante todo el camino.

A mi novia, Isabel, que me ha acompañado, cuidado y apoyado durante todo mi periodo en la universidad.

A mis amigos, que han amenizado mi estancia en la universidad. En especial a Daniel Morell y Adriana Iglesias, apoyos imprescindibles que me han ayudado y enseñado en todo momento.

A mi abuela, cuya mirada orgullosa me motivó a no rendirme y a confiar en mí mismo.

Por último, a los docentes de la Escuela Politécnica Superior que se esfuerzan en enseñar a sus alumnos, generarles interés, y prepararles para su futuro.

Resumen

La domótica consiste en la automatización del hogar. Los sistemas domóticos, son aquellos capaces de domotizar una vivienda; proporcionan servicios de comunicación, seguridad, eficiencia energética...etc. Sin duda alguna, la comunicación entre estos sistemas es algo esencial, existiendo redes cableadas e inalámbricas para ello. Estas comunicaciones nos permiten controlar nuestros sistemas domóticos sin la necesidad de encontrarnos en nuestro hogar.

BRIMO es un proyecto de código abierto para la gestión y el control de dispositivos domóticos en el hogar. Es una alternativa open source de bajo coste para todas aquellas personas que deseen domotizar su hogar de una manera barata y sencilla. No utiliza protocolos privados, y cualquier usuario puede utilizar y modificar la aplicación a su gusto.

Además, los usuarios pueden crear sus dispositivos (sensores, actuadores o cámaras) de manera sencilla, siempre que éstos sigan los requisitos establecidos. BRIMO nos ayuda, gracias a una interfaz sencilla e intuitiva, a ordenar nuestros dispositivos, visualizar sus estados y mandar comandos a los dispositivos que los acepten.

La aplicación está diseñada para ejecutarse en entornos ligeros, concretamente en una Raspberry Pi 3 (precio asqequible), pero también puede ser ejecutada en cualquier ordenador tras una simple configuración. Utiliza una arquitectura REST sobre el protocolo HTTPS para comunicarse con los dispositivos.

La función principal de BRIMO es de "bridge": punto común entre el usuario y los dispositivos. Se encarga de poner en contacto al usuario con los dispositivos.

Palabras clave

Domótica, código abierto, REST, raspberry, HTTPS, sensores, API, actuadores, bridge.

Abstract

Domotic consists of home automation. Domotic systems are those capable of automating a home; providing communication services, security services, energy efficiency services... etc. Communication between these systems is essential, existing wired and wireless networks for it, that allows us to controll them from inside and outside the home.

BRIMO is an open-source project for the management and control of domotic devices in the home. It is a low-cost open source alternative for all people who want to domotize their home in a cheap and simple way. It does not use private protocols, and anyone can use or modify the application on its preferences.

Besides, users are able to create their own devices (sensors, actuators or cameras) in a simple way, following the application requirements. Brimo helps us, thanks to a simple and intuitive interface, to arrange our devices, see their status and send them commands.

The application is designed to be runned on lightweight devices, specifically into a Raspberry Pi 3 (low cost), but it could be also runned on any computer after a simple configuration. It uses REST architecture over HTTPS protocol to communicate with devices.

The main function of Brimo is to act as bridge, the common point between users and devices: Brimo is the responsible of the communication between them.

Keywords

Domotic, open-source, REST, raspberry, HTTPS, sensors, API, actuators, bridge.

Índice

Listas

Lista de ecuaciones

Lista de figuras

Lista de tablas

1

Introducción

1.1. Motivación del proyecto

Los sistemas domóticos, por lo general, utilizan una arquitectura centralizada: un controlador (bridge) es el encargado de enviar y recibir información entre dispositivos domóticos e interfaces. Se utilizan sistemas centralizados porque abaratan mucho el coste de los dispositivos domóticos, que se fabrican con poca electrónica y programación, ya que la responsabilidad principal reside en el bridge. Este enfoque tiene sentido cuando existen muchos dispositivos en un hogar, que es el caso ideal. Si solo tuviésemos un dispositivo carecería de sentido tener un sensor y un bridge para manejarlo.

El problema principal que existe con los sistemas centralizados se encuentra en la **compatibilidad** entre dispositivos y bridges. Por lo que he observado, todavía falta mucha estandarización en el ámbito de la domótica: cada fabricante usa sus medios y protocolos haciendo incompatibles bridges y dispositivos. Además, estos dispositivos no suelen ser muy asequibles. Por lo tanto, nos encontramos ante la necesidad de comprar todos los dispositivos de una misma marca o a manejar cada dispositivo desde su correspondiente bridge, lo que nos obligaría a tener muchos bridges.

La domótica puede hacernos la vida en el hogar mucho más sencilla, ayudándonos a ahorrar tiempo, consumo de energía y costes que podremos invertir en otras cosas. Los hogares todavía están muy poco automatizados, y mi principal motivación ha sido acercar la domótica a las personas y aprender acerca de ella. Gracias al sistema planteado, manejamos todos los dispositvos a través de un solo bridge de manera sencilla y eficaz.

1.2. Objetivos

El objetivo de mi proyecto es desarrollar un sistema que sea capaz de recibir y enviar información entre dispositivos domóticos y usuarios. El usuario tiene que poder ver la información actualizada de sus dispositivos, enviarles comandos y ser capaz de gestionarlos:

- Asignarles un nombre: el usuario tiene que ser capaz de asignar un nombre a sus dispositivos.

- Eliminar dispositivos: el usuario tiene que ser capaz de eliminar dispositivos.
- Asignarles una ubicación: el usuario debe ser capaz de asignar una ubicación o habitación a sus dispositivos.

Además, el sistema debe ser ligero, ya que está pensado para poder ejecutarse en una raspberry pi. La ejecución en una raspberry pi nos permite dejar el sistema activo las 24 horas con un coste energético bajo.

1.3. Metodología y plan de trabajo

Debido a que muchos de los capítulos del documento coinciden con las fases de ciclo de vida del proyecto, es importante explicar la metodología llevada a cabo antes de explicar la estructura del documento.

La metodología que se ha utilizado para el desarrollo del proyecto es una metodología tradicional con un ciclo de vida en cascada. Se utilizarán las siguientes etapas, que se realizarán una detrás de otra (en cascada):

- Requisitos: en esta fase se analizarán los requisitos y las necesidades del sistema a desarrollar. Un buen análisis guía un buen desarrollo, mientras que un análisis mal hecho puede significar que el proyecto tenga retrasos, fallos e incluso su cancelación.
- Diseño: se detallarán la arquitectura, los componentes y los subsistemas que formarán parte del sistema final, así como la interacción entre ellos.
- Implementación: esta es la primera fase en la que se comienza a materializar el sistema. Se definirán las tecnologías a utilizar y se llevará a cabo el desarrollo de un sistema que siga el diseño realizado y cumpla los requisitos descritos.
- Verificación: también conocida como fase de pruebas, en esta fase se probará que el sistema cumpla con los requisitos especificados. Se utiliza para detectar posibles errores.
- Mantenimiento: durante esta fase se implanta el sistema y se atienden posibles incidencias que puedan surgir a lo largo del uso del mismo.

1.4. Estructura del documento

A lo largo del siguiente documento se irán explicando todas las fases del ciclo de vida del proyecto, además de tener una fase previa de análisis del estado del arte y una fase posterior de trabajos futuros y conclusiones:

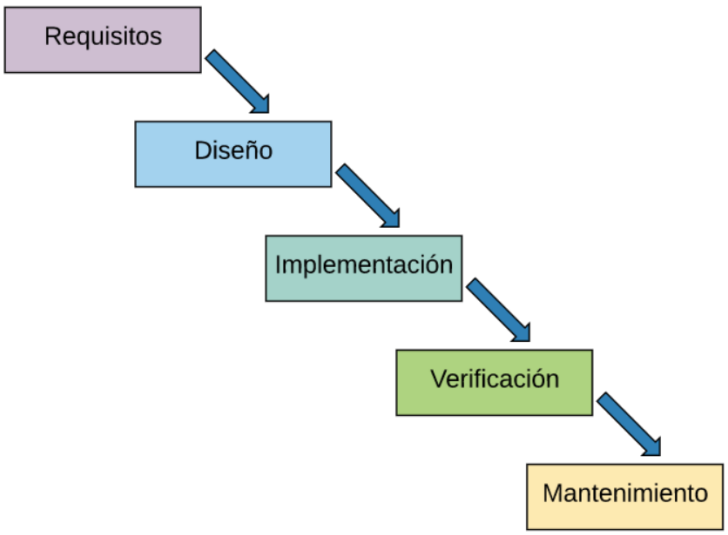


Figura 1.1: Las etapas del modelo en cascada. Recuperado de: https://openclassrooms.com/en/courses/4309151-gestiona-tu-proyecto-de-desarrollo/4538221-en-que-consiste-el-modelo-en-cascada

- **Estado del arte - Capítulo 2**: breve introducción y análisis de la domótica en la actualidad.
- **Análisis y diseño - Capítulo 3**: a lo largo de este capítulo analizan los requisitos de la aplicación y se diseña la arquitectura del sistema.
- **Desarrollo del sistema - Capítulo 4**: en este capítulo se describirán las tecnologías utilizadas para el desarrollo del sistema, y se detallarán algunos aspectos técnicos acerca de la implementación.
- **Pruebas - Capítulo 5**: en este capítulo se detallarán las pruebas realizadas para comprobar el correcto funcionamiento del sistema y la detección de posibles bugs.
- **Trabajos futuros y conclusiones - Capítulo 6**: análisis de posibles trabajos futuros para el HUB y conclusiones obtenidas a lo largo del trabajo.

2

Estado del arte

2.1. Introducción

La domótica, ***domus*** (casa en latín) ***autónomo*** (autogobernado en griego), se refiere a todos los sistemas que son capaces de automatizar un hogar (o cualquier otro edificio). Actualmente existen multitud de dispositivos domóticos, fabricantes y protocolos de comunicación. Cerraduras inteligentes, cámaras de seguridad, altavoces o termostatos son algunos de los dispositivos que nos podemos encontrar en la actualidad.

La automatización del hogar nos puede ayudar con tareas del día a día, y tienen multitud de aplicaciones. Explicaremos algunas aplicaciones prácticas de la domótica en la siguiente sección, y se hará un breve resumen de la domótica en la actualidad para terminar con el capítulo.

2.2. Aplicaciones

La domótica tiene infinidad de aplicaciones que nos pueden ayudar en nuestro hogar. Algunos de los ámbitos principales en los que la domótica nos puede ser de gran ayuda son:

- Seguridad: la domótica nos puede ayudar a mantener nuestro hogar seguro a través de diferentes sistemas. Por ejemplo: Las alarmas antirrobo automatizan la acción de llamar a la policía si un intruso es detectado por un sensor, las alarmas antifuego son capaces de detectar humo, hacer sonar sirenas e incluso ayudar a la extinción del fuego mediante la expulsión de agua.
- Eficiencia energética: los sistemas domóticos nos pueden facilitar el ahorro energético en nuestros hogares. Es una de las aplicaciones más utilizadas, sobre todo en grandes edificios (hoteles, fábricas, hospitales...), ya que el ahorro energético supone también un ahorro económico. Como ejemplos de esta aplicación tenemos: luces encendidas a través de sensores de movimiento, termostatos automatizados que regulan la temperatura en función de la temperatura ambiente y la franja horaria o enchufes inteligentes que encienden o apagan dispositivos según las necesidades pre programadas.

- Comodidad: la domótica puede ahorrarnos mucho tiempo, y hacer nuestro hogar más confortable, automatizando acciones cotidianas. Ejemplos de estas acciones son: apagado de luces, programación de alarmas, control de dispositivos multimedia, etc.
- Accesibilidad: esta aplicación de la domótica ayuda a las personas con limitaciones a tener más autonomía y seguridad en su día a día. Ya existen dispositivos como la teleasistencia de la Cruz Roja [1], que permiten a las personas avisar a emergencias tan sólo pulsando un botón. Además, existen sistemas de vigilancia remota, timbres lumínicos para personas con problemas de audición...etc.

2.3. Instalaciones domóticas

Existen instalaciones domóticas de tres tipos:

- Cableadas: los dispositivos se comunican entre ellos por cable. Una instalación cableada requiere un gran coste, y en edificios antiguos son menos factibles, sin embargo, son instalaciones más fiables.
- Inalámbricas: los dispositivos se comunican entre sí utilizando medios inalámbricos como WiFi, Bluetooth o infrarrojos.
- Mixtas: se trata de una instalación en la que existen tanto dispositivos cableados como dispositivos inalámbricos. Debido a la diversidad de medios y protocolos utilizados en estas instalaciones, se requiere un “bridge“ que gestione todas estas conexiones.

Además, existen diferentes arquitecturas de sistemas domóticos:

- Centralizados: en estas arquitecturas existe un dispositivo central, también conocido como **bridge**, que es el encargado de comunicarse con el resto de dispositivos: actuadores y sensores. La ventaja de estas arquitecturas es que pueden albergar dispositivos que utilicen diferentes protocolos y medios físicos de comunicación (instalación mixta). Un inconveniente de este tipo de arquitectura es la necesidad del propio bridge: conlleva costes elevados y su rotura o desconexión supone la caída total del sistema.
- Descentralizados: en esta arquitectura existen varios dispositivos centrales. Los dispositivos se conectan a estos dispsoitivos centrales, que pueden ser de diferentes tipos. Estos dispositivos son más complejos, pero también son más ampliables, ya que si existe un número elevado de dispositivos conectados a un solo bridge, la arquitectura centralizada puede colapsarse.
- Distribuidos: en esta arquitectura todos los dispositivos son a su vez controladores. Son capaces de enviar la información al sistema y actuar en consecuencia a información recibida de otros

dispositivos. Estos sistemas son más fiables, ya que la desconexión de un dispositivo no significa la caída del sistema, pero son más complejos y requieren más programación.

2.4. Comunicación en sistemas domóticos

Existen multitud de protocolos para la comunicación entre dispositivos domóticos, y podemos diferenciar entre protocolos para instalaciones cableadas o protocolos para instalaciones inalámbricas.

2.4.1. Comunicación por cable

Los sistemas domóticos que se comunican por cable suelen utilizar, por lo general, estos tipos de conexiones:

- Cable BUS (KNX): este tipo de cable BUS es utilizado exclusivamente por los dispositivos de la instalación domótica. Estas instalaciones son costosas, pero son muy fiables: la instalación no está expuesta a interferencias, saturación...etc.
- Cable PLC(X10): este tipo de conexión utiliza el cableado de alimentación doméstico para la comunicación entre los dispositivos. No se requiere instalación adicional, ya que los hogares se fabrican con instalación eléctrica, pero es una conexión muy expuesta a interferencias. A pesar de ser fácil de instalar, no es recomendable, a excepción de no poder utilizar otros medios.

2.4.2. Comunicación inalámbrica

En la comunicación inalámbrica, sin embargo, existen multitud de tipos de conexión. Las más concoidas son, comunicación via Wi-Fi, Bluetooth, Infrarrojos...etc.

No obstante, existen protocolos destinados exclusivamente a la Domótica. Los más famosos, y de los que hablaremos en las siguientes subsecciones son: Insteon, Z-Wave y ZigBee.

Insteon

Insteon es una tecnología orientada a la domótica. Está orientada a dispositivos conectados a la corriente eléctrica y utiliza la red eléctrica y/o la radio para establecer la comunicación entre ellos.

La tecnología Insteon utiliza una arquitectura “peer to peer” en la que todos los dispositivos sirven como repetidores, repitiendo los mensajes recibidos, de forma que cuantos más nodos haya más alta es la probabilidad de que el nodo destino reciba el mensaje.

En la arquitectura INSTEON existen tres tipos de dispositivos:

- RF-only: comunicación vía radio únicamente.
- Powerline-only: comunicación vía PLC únicamente.
- Dual-Band: comunicación PLC y radio.

2.4.3. Z-Wave

Z-Wave es un protocolo para la comunicación inalámbrica orientado a la domótica. Z-Wave trabaja en el rango de frecuencias MHZ, a diferencia de Wi-Fi y otros sistemas, que operan en frecuencias de 2.4 GHz. De esta manera, Z-Wave evita interferencias y hace que sus conexiones sean más fiables.

Z-Wave está orientada al bajo consumo, y sus dispositivos pueden permanecer en modo ahorro hasta el 99 % del tiempo total. Esto hace que Z-Wave sea la tecnología ideal para dispositivos que operan con batería.

En los sistemas Z-Wave existen dos tipos de dispositivos:

- Controladores: son los encargados de enviar comandos e inicar la comunicación con los diferentes esclavos.
- Esclavos: reciben los comandos por parte de los controladores y actúan en consecuencia.

ZigBee

ZigBee es un mecanismo de conexión inalámbrica (como Bluetooth o WiFi) que se caracteriza por su bajo consumo energético, y está orientado a la domotización. Al contrario que en Bluetooth los dispositivos se encuentran “dormidos” la mayor parte del tiempo, ya que está destinado a dispositivos como bombillas, persianas, detectores de humo...etc, que no requieren una comunicación constante.

Un inconveniente de ZigBee es que no puede conectarse directamente con móviles u otros dispositivos, ya que usualmente estos dispositivos no tienen esta tecnología, al contrario que con Bluetooth y WiFi. Por lo tanto, es necesario un “bridge” para poder controlar los dispositivos ZigBee desde el resto de dispositivos (móviles, tablets, ordenadores...etc). Otro inconveniente de la tecnología ZigBee es su velocidad de transmisión, de 250 kbits/s frente a los 32 Mb/s de Bluetooth, lo que limita a ZigBee a dispositivos destinados a domótica y lo excluye de dispositivos de audio, cámaras, móviles...etc.

3

Análisis y diseño del sistema

3.1. Necesidades del sistema

En esta sección se analizan las necesidades los dispositivos con los que nos comunicaremos y requisitos del sistema.

3.1.1. Necesidades de los dispositivos

Antes de empezar a diseñar el sistema, elegir el protocolo que se utilizará y el medio físico por el que se comunicarán los dispositivos, es necesario analizar los dispositivos que podrán conectarse al HUB, así como sus necesidades. Una vez determinados los requisitos del protocolo se estudiará el medio físico de comunicación.

Los principales dispositivos domóticos que he encontrado son: sensores de temperatura, sensores de humedad, sensores de luz, sensores de movimiento, medidores de distancia, sensores de humo, sensores magnéticos, cámaras, bombillas, enchufes, termostatos, motores, aires acondicionados, interruptores y altavoces. Estos dispositivos pueden ser divididos en dos grupos: **sensores y actuadores**.

Los sensores solamente enviarán información al HUB (comunicación unidireccional), mientras que los actuadores, además de enviar el estado en el que se encuentran, recibirán mensajes con diferentes comandos (comunicación bidireccional).

3.1.2. Requisitos del sistema

Una vez analizadas las necesidades de los dispositivos podemos analizar las necesidades del sistema. Para ello diferenciaremos entre requisitos funcionales y requisitos no funcionales:

- Requisitos funcionales: son aquellos que especifican funcionalidades propias del sistema.
- Requisitos no funcionales: este tipo de requisitos no describen funciones a realizar por el sistema, sino necesidades del sistema para un correcto funcionamiento global. Por ejemplo: rendimiento

del sistema, estabilidad del sistema, seguridad...etc.

Requisitos funcionales

1. **Gestión de usuarios**

   1.1. **Acceso al sistema**: los usuarios podrán acceder al sistema mediante sus credenciales. El sistema no permitirá el acceso a usuarios sin credenciales de acceso correctas.

   1.2. **Restricción por roles**: las funcionalidades del sistema estarán restringidas según el rol del usuario autenticado.

   1.3. **Creación de nuevos usuarios**: los usuarios con rol administrador podrán crear nuevos usuarios y asignarles un rol.

   1.4. **Cambio de credenciales**: todos los usuarios podrán cambiar sus credenciales de acceso.

   1.5. **Edición de usuarios**: los usuarios administradores podrán administrar al resto de usuarios no administradores: editar sus credenciales y eliminarlos.

2. **Comunicación con los dispositivos**

   2.1. **Registro de dispositivos**: el sistema permitirá el registro en el sistema de nuevos dispositivos de manera automática. Tras el registro el sistema guardará la información del dispositivo. El sistema debe permitir el registro de dispositivos con cualquier tipo de comando, y no ceñirse a un número cerrado de comandos (ON/OFF, +/-,...). De esta manera cualquier actuador podrá conectarse al HUB, siempre y cuando los comandos sean registrados de manera correcta.

   2.2. **Recibir información de los dispositivos**: el sistema será capaz de recibir información de los dispositivos y actualizarla en su base de datos.

3. **Gestión de dispositivos**

   3.1. **Renombrar dispositivos:** los usuarios comunes y administradores serán capaces de cambiar el nombre de un dispositivo tantas veces como deseen.

   3.2. **Enviar comandos a los dispositivos:** los usuarios comunes y administradores podrán enviar comandos a los dispositivos y observar la respuesta recibida.

   3.3. **Modificar localización de los dispositivos:** los usuarios comunes y administradores serán capaces de modificar la localización de cualquier dispositivo.

   3.4. **Eliminación de dispositivos:** los usuarios comunes y administradores podrán eliminar dispositivos del sistema de manera permanente.

4. **Visualizar información de los dispositivos:** todos los usuarios con acceso al sistema serán capaces de observar los dispositivos dados de alta y su información actualizada. Los usuarios serán capaces de filtrar por ubicación.

5. **Gestión de ubicaciones**

   5.1. **Añadir ubicación:** los usuarios comunes y administradores serán capaces de añadir nuevas ubicaciones al sistema.

   5.2. **Editar ubicación:** los usuarios comunes y administradores serán capaces de editar el nombre de las ubicaciones existentes en el sistema.

   5.3. **Eliminar ubicaciones:** los usuarios comunes y administradores podrán eliminar ubicaciones del sistema, de tal manera que los dispositivos asignados a dichas ubicaciones pasarán a no tener ubicación asignada.

Requisitos no funcionales

1. **Seguridad:** el sistema debe ser seguro y no deben existir vulnerabilidades. Es imprescindible que toda comunicación se realice de manera segura, logrando que nadie pueda modificar o acceder a nuestra información.

2. **Escalabilidad:** aunque durante la realización del proyecto nos centraremos únicamente en la comunicación mediante protocolo HTTPS, es necesario diseñar un sistema escalable que en el futuro pueda funcionar con diferentes protocolos, tecnologías y dispositivos.

3. **Ligereza:** el sistema debe ser ligero y poder ejecutarse en una Raspberry Pi.

4. **Interfaz sencilla y adaptable a cualquier dispositivo:** la interfaz de usuario debe ser sencilla e intuitiva para los usuarios. Además, debe ser compatible con cualquier dispositivo móvil sin perder funcinalidad.

5. **Conexión a subred:** para el correcto funcionamiento del sistema necesitamos que todas las partes que lo integran (HUB, dispositivos domóticos y dispositivo móvil del usuario) formen parte de la misma subred. No es necesario que dicha subred tenga conexión con el exterior.

3.2. Casos de uso

Para ayudarnos a diseñar el software, es de gran utilidad un diagrama de casos de uso, en el que se describen todas las acciones que el usuario puede llevar a cabo.

Además, el diagrama de casos de uso será de gran utilidad a la hora de diseñar la interfaz de nuestra aplicación, ya que para que la interfaz sea válida, es necesario que se cumplan todos los casos de uso.

3.2.1. Actores

Debido a que el usuario podrá interactuar totalmente con el sistema y podrá llevar a cabo acciones irreversibles (como borrar un dispositivo), es necesario que el sistema cuente con funcionalidad para la gestión de usuarios basada en roles, de tal manera que los usuarios puedan llevar a cabo sólo las acciones que su rol les permita.

Distinguiremos tres roles de usuario, y por lo tanto, tres actores en nuestro sistema:

- **Usuario lurker:** este usuario sólo es capaz de ver el estado actual de los diferentes dispositivos, así como su localización.
- **Usuario común:** este usuario, además de ver el estado y la localización de los dispositivos, puede enviar comandos a los actuadores y gestionar los dispositivos: eliminarlos, modificar su localización...etc.
- **Usuario administrador:** la única diferencia de este usuario con el usuario común es que este usuario es capaz de gestionar usuarios: dar de alta nuevos usuarios, cambiar el rol de los usuarios y eliminar usuarios.

3.2.2. Diagrama de casos de uso

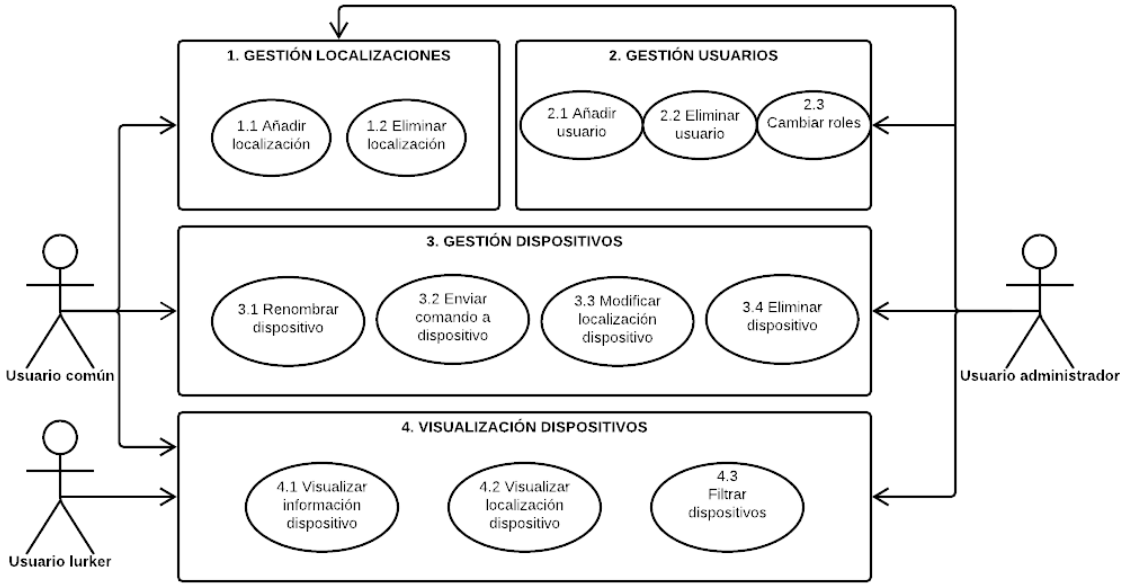


Figura 3.1: Diagrama de casos de uso

3.3. Diagramas de secuencia

Como hemos descrito anteriormente, en nuestro sistema participarán diferentes “partes“ o subsistemas, por lo que es necesario definir la interacción entre cada una de ellas. Así como en el diagrama de casos de uso hemos descrito cómo el usuario interactuará con el sistema, en los diagramas de secuencia describiremos cómo interactuarán los subsistemas entre sí.

Debido a que en la mayoría de las secuencias solamente interactúan el HUB y la interfaz, se realizarán dos diagramas de secuencia: uno común para todas las acciones en las que participan HUB e interfaz, y otro con las secuencias en los que es necesaria la interacción de todos los subsistemas.

3.3.1. Subsistemas

Los subsistemas que participarán en los diagramas de secuencia son:

- **HUB**: es la parte central o bridge del sistema. Se encarga de comunicarse con los dispositivos y con la interfaz de usuario. Única ejecución, contiene también el servidor de datos.
- **Interfaz:** interfaz a través de la cual el usuario podrá interactuar con el HUB y los dispositivos. Se ejecuta en el dispositivo móvil del usuario, por lo que existirá una ejecución por cada usuario utilizando la aplicación.
- **Dispositivos:** son los encargados de informar al usuario de su estado y, en el caso de los actuadores, ejecutar una acción a partir de un comando.

3.3.2. Diagrama de secuencia Usuario-Interfaz-HUB

Este flujo es el que seguirán todos los casos de uso a excepción de los casos **3.2 Enviar comando a dispositivo** y **4.1 Visualizar información dispositivo**. Es el flujo habitual que siguen los frameworks web que utilizan el modelo MVC:

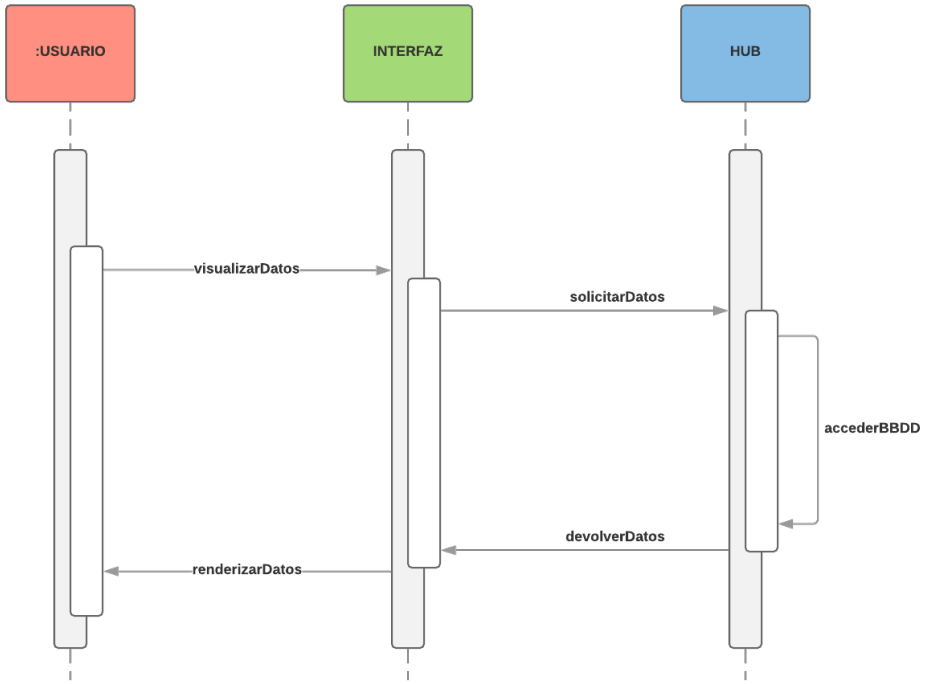


Figura 3.2: Diagrama de secuencia Usuario-Interfaz-HUB

3.3.3. Diagrama secuencia Usuario-Interfaz-HUB-Dispositivos

En este diagrama se explican dos flujos:

- **4.1 Visualizar información dispositivo:** la interfaz, por defecto, muestra la información actualizada de los dispositivos. Los dispositivos mandan su información al HUB de manera constante, y la interfaz se encarga de pedir el estado de los dispositivos al HUB.
- **3.2 Enviar comando a dispositivo:** un usuario desea enviar un comando a un dispositivo. La interfaz envía el comando al HUB, que es el encargado de enviárselo al dispositivo.

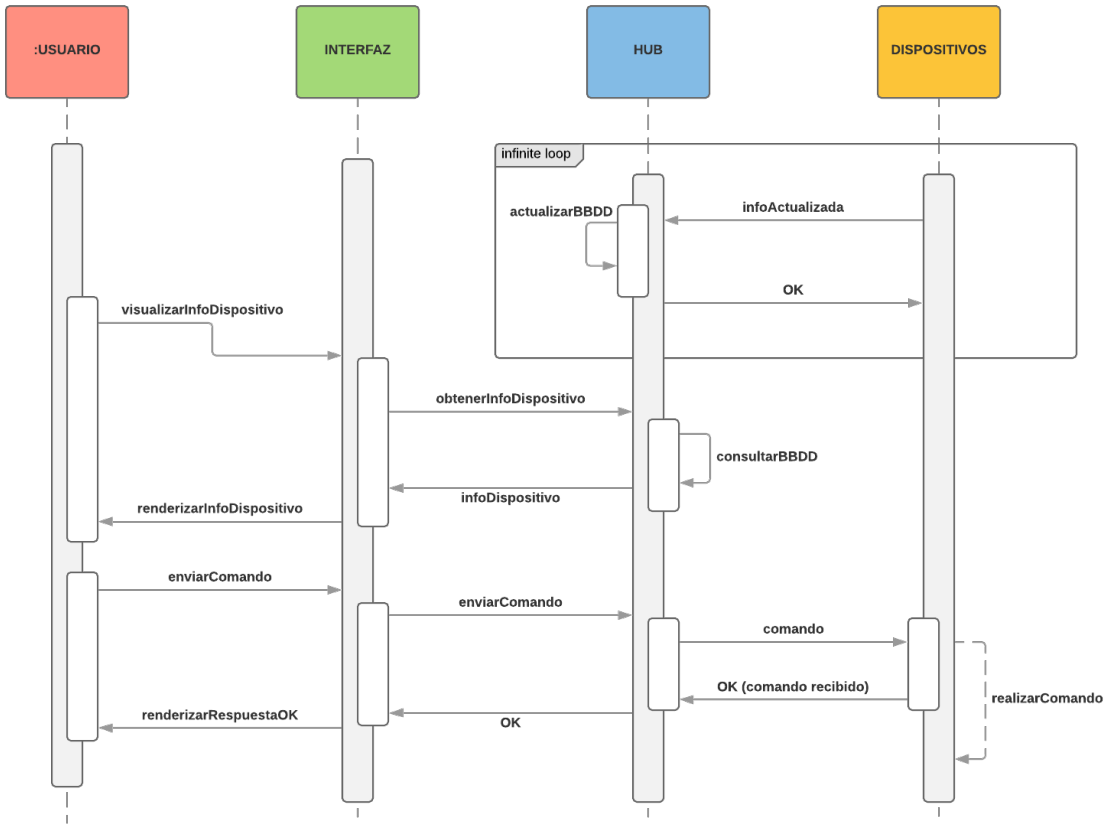


Figura 3.3: Diagrama de secuencia Usuario-Interfaz-HUB-Dispositivos

3.4. Protocolo

En esta sección se describirá el protocolo que se utilizará en las comunicaciones entre los dispositivos y el HUB.

3.4.1. Protocolo HTTPS

El protocolo elegido para la comunicación entre dispositivos HUB e interfaz es el protocolo HTTPS (Hypertext Transfer Protocol Secure).

Este protocolo nos ofrece la posibilidad de implementar una API REST consumible por parte de los dispositivos y por parte de la interfaz, sin necesidad de utilizar distintos protocolos en cada caso.

Una API REST proporciona una interfaz o contrato entre diferentes sistemas que utilicen HTTP/S como medio comunicación. Las APIs REST están muy estandarizadas a día de hoy, y nos dan la capacidad de separar lógica y funcionalidad entre cliente y servidor, y de ser capaces de utilizar diferentes lenguajes y tecnologías en cada una de las partes. Es decir, podemos tener un servidor escrito en Express.js (JavaScript), una interfaz gráfica utilizando Angular5 (TypeScript), y unos actuadores/sensores que utilicen CherryPy (Python).

Además, utilizar HTTPS nos ofrece la posibilidad de crear un canal de comunicación cifrado, de manera que la información que circula en dicho canal no pueda ser descifrada por ningún intermediario ni se pueda sufrir un ataque Man-In-The-Middle*.

3.5. Arquitectura del sistema

En esta sección se describe el diseño y la arquitectura del sistema de manera global, incluyendo dispositivos actuadores, dispositivos sensores y el propio HUB.

3.5.1. Arquitectura del sistema

Teniendo en cuenta las necesidades de nuestro sistema realizaremos una arquitectura similar a las arquitecturas de microservicios, en la que el HUB y los actuadores serán hosts de un servidor REST y serán capaces de recibir y procesar peticiones. Esto requerirá n + 1 servidores REST, siendo n el número de dispositivos actuadores.

El HUB recibirá peticiones de parte de la interfaz de usuario y de los dispositivos, y lanzará peticiones a los actuadores. Para ello se establecerán dos APIs que serán publicadas por el HUB y consumidas por la interfaz de usuario y los dispositivos:

- Interface API: será consumida por la interfaz de usuario. Se encargará de enviar la información de los dispositivos al usuario: número de dispositivos, información, localización, etc... Además, permitirá al usuario gestionar dispositivos y enviarles comandos.

- Sensors API: será consumida por actuadores y sensores por igual. Permitirá a los dispositivos darse de alta en el sistema y actualizar periódicamente su información.

Los actuadores, además de lanzar peticiones al HUB para informar de su estado, deberán ser capaces de recibir peticiones del HUB con diferentes comandos. Para ello se establecerá otra API que todos los dispositivos deberán seguir, para que así el HUB consuma la misma API en los diferentes dispositivos. Será denominada en adelante como Actuators API. Estableceremos y explicaremos estas tres APIs en la Subsección 3.5.3.

Como mencionamos anteriormente, la seguridad es un requisito indispensable de nuestro sistema, y debemos asegurar que ningún intruso pueda acceder y/o modificar nuestra información. Para cumplir nuestro requisito, todas las peticiones HTTPS serán securizadas, asegurando así una comunicación segura entre los diferentes servidores.

Esquema de la arquitecura a seguir:

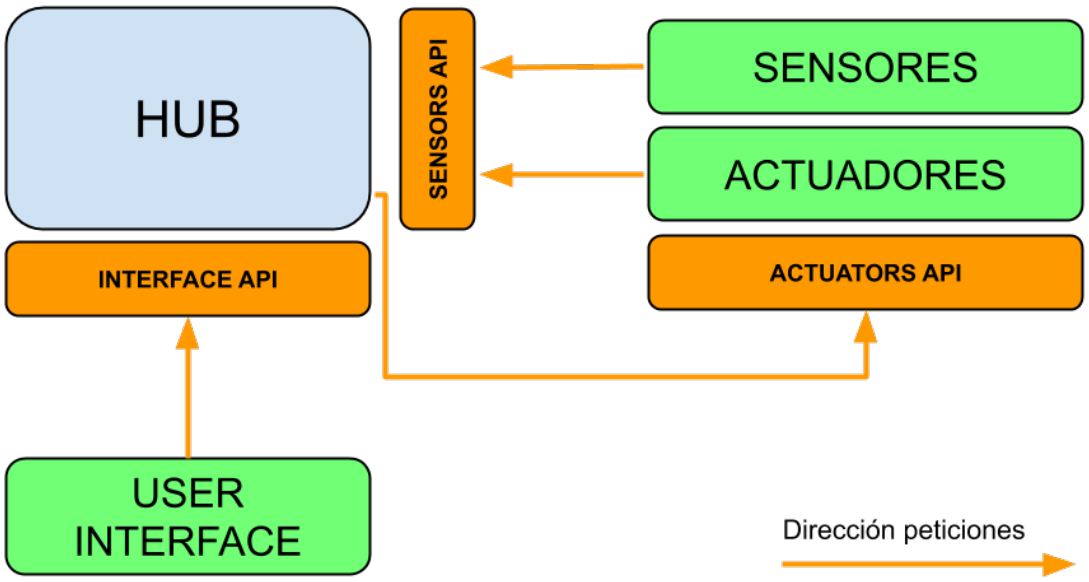


Figura 3.4: Esquema de la arquitectura

3.5.2. Modelo de datos

En esta sección se describirán los modelos de datos a utilizar. Todos los datos residirán en el HUB, que será el encargado de orquestarlos, organizarlos y mantenerlos.

Analizando las necesidades del sistema nos encontramos con cuatro entidades a definir:

- Dispositivos: esta entidad se utilizará para almacenar la información de los actuadores/sensores. En el caso de los actuadores será necesario guardar su dirección IP para poder enviarles comandos.
- Comandos: entidad para almacenar los comandos de los diferentes dispositivos. Cada comando tendrá un código y una descripción.
- Habitaciones: esta entidad nace de la necesidad de organizar los dispositivos de una casa en grupos más pequeños. Una manera lógica y muy común es por habitaciones, cada dispositivo podrá o no pertenecer a una habitación.
- Usuarios: es necesario restringir los usuarios que pueden tener acceso a nuestro sistema. Las credenciales de cada usuario se guardarán en esta tabla, así como su rol.

Diagrama entidad-relación modelo de datos:

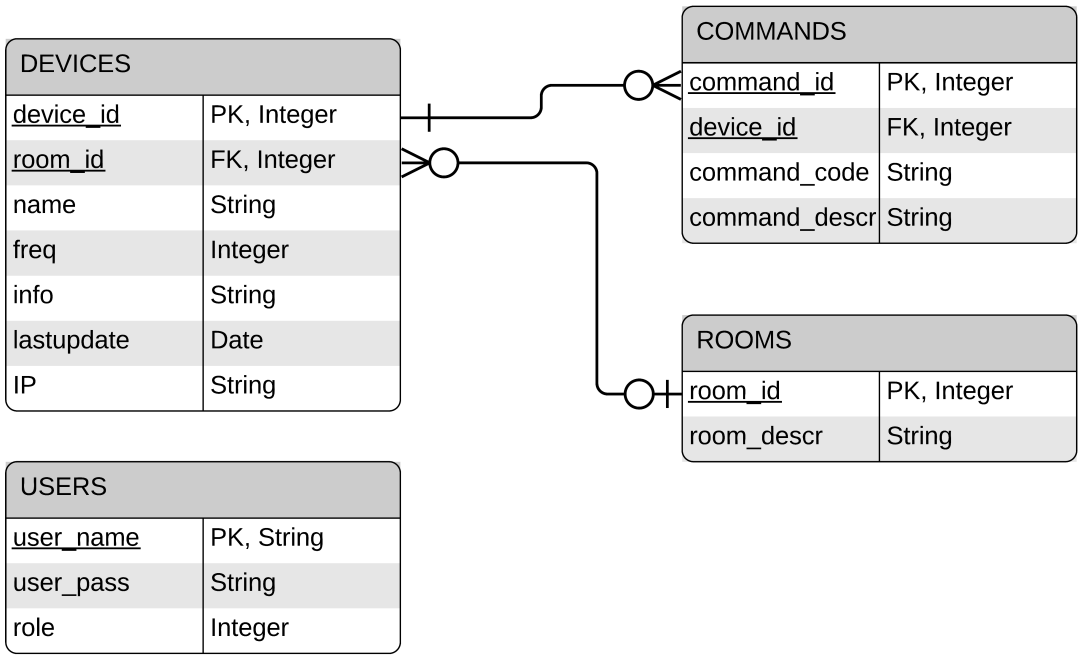


Figura 3.5: Diagrama ER

3.5.3. APIs

Una vez numeradas las diferentes APIs a utilizar y definido el modelo de datos podemos comenzar a definir detalladamente cada una de las APIs. Al tratarse de APIs REST, en la definición de cada método es necesario informar: ruta del método, verbo (GET, POST, PUT, PATCH, DELETE), cuerpo de la petición (si existiera) y variables de la ruta (si existieran).

A no ser que se indique lo contrario, el cuerpo de todas las peticiones debe estar en formato JSON, y debe ser informada la cabecera **Content-Type** con valor **application/json**.

Como se ha descrito anteriormente todas las peticiones deberán ir securizadas, para lo que se utilizarán tokens JWT. Es imprescindible que el token vaya en la cabecera **x-access-token** de cada petición, o de lo contrario la petición será denegada. La generación de tokens y la gestión de usuarios es descrita por la API de login.

Estas APIs funcionan como contratos entre publicador y consumidor, y es imprescindible que ambas partes consuman y publiquen de la manera acordada para que el sistema completo funcione. El cambio de uno de estos contratos debe ser indicado a todas las partes para que se tenga en cuenta en los desarrollos futuros.

Todos los métodos de estas APIs estarán enumerados y definidos en un proyecto de Postman desde el cual se podrán probar.

Sensors API

Esta API será consumida tanto por sensores como por actuadores, y les permitirá registrarse en el sistema y actualizar su información.

1. **Registro de dispositivos:** se utilizará para el registro de dispositivos. En el cuerpo de la petición se informarán un nombre descriptivo (podrá ser modificado por el usuario más adelante) y frecuencia de actualización de la información [1]. Opcionalmente, en el caso de ser un actuador, el dispositivo informará de los comandos que es capaz de recibir. Estos comandos vendrán en forma de array y deben contener descripción y código de comando. Si el registro es correcto el HUB responderá con un 201 CREATED y un id de dispositivo. Este id será utilizado por el dispositivo más adelante para enviar información al HUB.
   Ruta: *POST /brimo/sensors-api/devices*

2. **Actualizar información dispositivo:** se utilizará para actualizar la información del dispositivo. El parámetro device-id indicará el id del dispositivo, proveniente del registro. Si la información se actualiza correctamente el HUB devolverá 200 OK. Una vez actualizada la información del

[1] Si el dispositivo pasa más de los segundos informados sin actualizar información, entonces el HUB lo considerará desconectado.

dispositivo se actualizará la hora de última actualización **(lastupdate)**.
Ruta: *PUT /brimo/sensors-api/devices/{device-id}/info*

Actuators API

Esta API será consumida por el HUB, y permitirá al HUB enviar comandos a los dispositivos.

1. **Enviar comandos:** es el único método de la API. El HUB enviará esta petición para enviar comandos al dispositivo. El código de comando debe haber sido informado previamente en la fase de registro del actuador.
   Ruta: *POST /brimo/actuators-api/commands?command_code=ON*

Interface API

Como hemos explicado anteriormente, esta API será consumida por la interfaz de usuario y permitirá al usuario obtener información de los dispositivos, gestionarlos y mandarles comandos:

1. **Obtener información de los dispositivos:** esta petición nos devolverá información de todos los dispositivos dados de alta en el sistema. Al tratarse de una lista no se poblarán todos los campos del dispositivo, sólo los comunes: id del dispositivo, nombre, frecuencia, fecha de última actualización de la información, id de habitación y descripción de la habitación.
   Ruta: *GET /brimo/interface-api/devices*
2. **Obtener información completa de un dispositivo:** a diferencia de la petición anterior, se obtiene únicamente la información del dispositivo indicado con el parámetro device-id. Esta información es más completa, y además de la información de la petición anterior se obtiene la lista de comandos que acepta el dispositivo y su IP.
   Ruta: *GET /brimo/interface-api/devices/{device-id}*
3. **Eliminar dispositivo:** a través de esta petición el usuario podrá eliminar el dispositivo indicado. A partir de este momento el sistema denegará al dispositivo la comunicación con el mismo.
   Ruta: *DELETE /brimo/interface-api/devices/{device-id}*
4. **Editar dispositivo:** esta petición permitirá editar la habitación en la que se encuentra el dispositivo y su nombre. Ambos parámetros room-id y name son opcionales, aunque al menos uno debe estar presente.
   Ruta: *PATCH /brimo/interface-api/devices/{device-id}/?room-id=12&name=sensor-habitacion*
5. **Enviar comando al dispositivo:** se utilizará para enviar comandos al dispositivo informado. La petición irá al HUB, que será el encargado de enviar otra solicitud al dispositivo correspondiente. Para ello, utilizará la IP del dispositivo.
   Ruta: *POST /brimo/interface-api/devices/{device-id}/commands?command-code=ON*

6. **Obtener habitaciones:** devolverá la lista actual de habtiaciones registradas. Se devolverán en forma de array y en cada una de ellas vendrán informadas descripción e identificador.
   Ruta: *GET /brimo/interface-api/devices/rooms*

7. **Añadir nueva habitación:** se utilizará por el usuario para añadir habitaciones. Únicamente es necesario informar el nombre de la nueva habitación. Si el registro de la habitación es correcto, entonces el HUB devolverá 201 CREATED con el id de la nueva habitación.
   Ruta: *POST /brimo/interface-api/devices/rooms*

Login API

Esta API será utilizada para la generación de los JWT, que necesariamente, deben ir informados en las cabeceras de cada petición. Además, gestionará los usuarios con acceso al sistema.

1. **Login**: se utilizará para validar las credenciales del usuario antes de acceder al sistema. Se deberán informar los campos usuario y contraseña para la correcta generación del token. En el caso de introducir credenciales inválidas el HUB devolverá 401 Unauthorized. En caso de éxito el HUB devolverá el token generado, que será válido para las siguientes dos horas.
   Ruta: *POST /brimo/login-api*

2. **Añadir usuarios**: servirá para añadir nuevos usuarios al sistema. Deberán informarse nombre de usuario y contraseña.
   Ruta: *POST /brimo/login-api/users*

3. **Editar usuarios**: petición para modificar la contraseña y/o el nombre de usuario actuales. Deberán ir informados en el cuerpo de la petición al menos uno de los dos parámetros.
   Ruta: *PUT /brimo/login-api/users*

4

Desarrollo del sistema

En este capítulo se explican las tecnologías utilizadas y las arquitecturas internas de cada subsistema. Los dos subsistemas que desarrollaremos serán el HUB y la Interfaz gráfica.

4.1. Stack tecnológico

Elegir el stack tecnológico a utilizar en el desarrollo de cualquier sistema es siempre una tarea difícil y determinante en el desarrollo de cualquier proyecto. Una elección inacertada puede significar el fracaso de un proyecto, mientras que una elección acertada sginificará que el proyecto salga adelante cumpliendo con todas las expectativas.

Para elegir el stack correcto necesitamos tener en cuenta varios aspectos:

- Madurez de la tecnología: es importante utilizar una tecnología con cierta madurez para evitar bugs, incompatibilidades, etc. Además, es conveniente optar siempre por versiones LTS y evitar versiones Beta.

- Comunidad/respaldo de la tecnología: por lo general, este punto está muy relacionado con el punto anterior, ya que cuanta más madurez tiene una tecnología, más comunidad suele existir. Aunque no sólo la madurez influye, también la popularidad de la tecnología es importante, cuanto más se use esa tecnología, más comunidad tiene. La comunidad de una tecnología, junto con su documentación, ayudan al desarrollador a enfrentarse a los problemas que surgen a lo largo del desarrollo, siendo capaz de ver o preguntar a otros desarrolladores que ya se hayan enfrentado a dichos problemas.

- Conocimiento de los desarrolladores: este punto es esencial, ya que es necesario que el equipo de desarrollo conozca las tecnologías, o en caso contrario, ser capaces de contratar nuevos desarrolladores que sí las conozcan. La popularidad de la tecnología es esencial en este punto, ya que, por lo general, es más sencillo encontrar desarrolladores que conozcan tecnologías populares.

- Librerías externas: junto a la comunidad, las librerías externas ayudan a los desarrolladores a solucionar problemas que ya han sido resueltos por otros desarrolladores. Por ejemplo, librerías para el manejo de fechas, librerías para gestionar conexiones con bases de datos...etc.
- Licencias/mantenimiento de la tecnología: es necesario tener en cuenta las licencias y el mantenimiento de las tecnologías antes de elegirlas, pues pueden suponer gastos bastante elevados.

Teniendo en cuenta los apartados anteriores, se ha optado por utilizar Express.js junto Sqlite para el desarrollo del HUB, e Ionic para el desarollo de la interfaz gráfica.

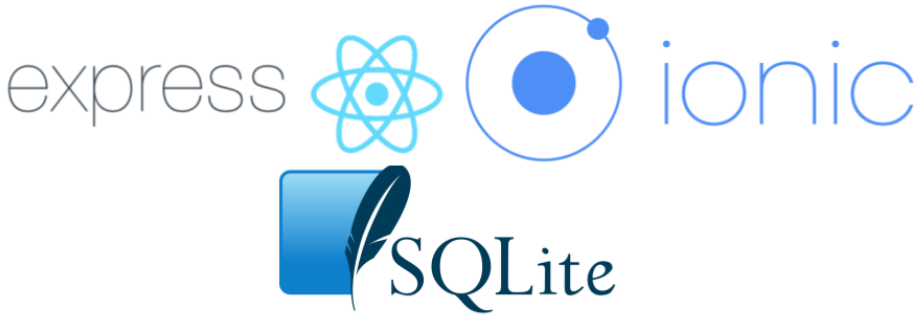


Figura 4.1: Stack tecnológico escogido

4.1.1. Express.js

Se trata de un framework open source para Node.js enfocado en el desarrollo de aplicaciones web. Utiliza JavaScript como lenguaje principal, un lenguaje moderno, muy popular y muy rápido en ejecución. Los desarrolladores de Express.js [2] definen a su framework como: *"Infraestructura web rápida, minimalista, y flexible para Node.js"*. Además, según hackr.io [3]: *"Express es uno de los frameworks que más rapido está creciendo en popularidad, y es utilizado por compañías como Accenture, IBM o Uber"*.

Al ser un framework para Node, podemos utilizar librerías de terceros de manera sencilla a través del gestor de paquetes NPM [4]. La comunidad de Node es una de las comunidades más activas y grandes que existen.

Según el Developer Survey 2018 [5] realizado por **StackOverFlow**, una de las comunidades de desarrolladores más grandes del mundo, JavaScript es la tecnología más popular (con un 71.5 %) dentro de las tecnologías de programación, scripting y lenguajes marcados.

Además, el mismo estudio revela que Node.js es la tecnología más popular (con un 49.9 %) dentro del apartado de frameworks, librerías y herramientas.

4.1.2. SQLite

SQLite [6] es una librería open source que implementa un pequeño gestor de bases de datos SQL. No necesita ejecutarse en un servidor a parte, lee y escribe directamente de un fichero. Es perfecto para nuestro caso, en el que no se necesitan muchas tablas y éstas no contendrán demasiados datos. Todo esto hace que SQLite se adapte muy bien a sistemas ligeros.

La no necesidad de ejecución en paralelo significa que apenas requiere configuración. Por otro lado, es una librería muy compacta y bastante madura, con inicios en el año 2000. Basándonos de nuevo en el **Developer Survey 2018** [5], SQLite es el quinto gestor de bases de datos preferido por los desarrolladores, con un 19.7 %, por delante de gestores muy conocidos como Redis, ElasticSearch, MariaDB u Oracle.

La madurez y popularidad de este gestor hace, como era de esperar, que tenga total compatibilidad con Node.js, y por lo tanto con Express, gracias al módulo NPM sqlite3 [7].

4.1.3. Ionic

Ionic [8] es un framework open source que nos permite construir aplicaciones híbridas de manera rápida y sencilla. Una aplicación híbrida es, realmente, una aplicación web que se ejecuta en un WebView dentro del dispositivo, aunque pueden tener acceso directamente a las APIs nativas del dispositivo: GPS, almacenamiento, contactos...etc.

La principal ventaja de Ionic es que el mismo desarrollo genera aplicaciones Android, IOS y Web, ya que se trata de aplicaciones web embebidas en un WebView. Además, Las aplicaciones Ionic están hechas por componentes reutilizables, e Ionic nos ofrece muchísimos componentes (“UI Components“) para construir la interfaz de nuestra aplicación de manera rápida: botones, alertas, menús, indicadores de carga...etc.

Ionic utiliza tecnologías Web para el desarrollo: TypeScript, HTML y CSS, lo que facilita mucho la migración de una aplicación móvil a una versión web y viceversa: la mayoría de servicios y lógica puede ser reutilizada, cambiando únicamente los componentes que se utilizan.

Al no tratarse de aplicaciones nativas, Ionic no está indicado para aplicaciones complejas, ya que el rendimiento y la gestión de recursos es mucho menos óptima en este tipo de aplicaciones. Sin embargo, no es el caso de nuestra aplicación, ya que necesitamos una aplicación sencilla, con una interfaz sencilla que pueda hacer peticiones fácilmente a una API REST, por lo que Ionic encaja a la perfección.

Además, Ionic ofrece “live reloading“: no es necesario recompilar o volver a ejecutar nuestra aplicación, los cambios son detectados en caliente y se actualiza automáticamente.

4.2. Hub

Como ya se ha explicado anteriormente, el HUB es la parte central del sistema. En él reside toda la lógica de negocio, y es el encargado de comunicarse con la interfaz y los dispositivos.

4.2.1. Módulos

En esta sección se definirán los módulos del HUB y las funciones que cumplen cada uno de ellos. La organización por módulos, además de permitirnos organizar el software de una manera clara, nos ayuda a construir software flexible: en futuras mejoras del HUB se podrán añadir nuevos módulos y funcionalidades con facilidad.

Por ejemplo, en el módulo de servicios reside toda la lógica interna, mientras que el módulo enrutador es el encargado de "traducir" los datos provenientes de la red a un modelo de datos conocido e invocar a los diferentes servicios. De esta forma, si en un trabajo futuro queremos añadir dispositivos bluetooth crearemos un módulo bluetooth que reutilice nuestros servicios.

Otro ejemplo sería la migración de la base de datos a otro motor diferente; sólo necesitaríamos cambiar el módulo repositorio, el resto del sistema se mantendría intacto.

Cada módulo debe ser independiente del resto, y la modificación interna de un módulo no debería requerir la modificación del resto de módulos.

En el Apéndice A pueden verse ejemplos del desarrollo de los diferentes módulos.

4.2.2. Módulo enrutador

Este módulo es el encargado de gestionar las conexiones entrantes y de manejar la información proveniente del exterior. Para nuestro caso, que utilizamos el protocolo HTTPS, en este módulo residirán las implementaciones de las APIS anteriormente definidas. Se encarga de implementar todas las rutas, encapsular los diferentes parámetros en objetos de nuestro modelo y enviar las respuestas y códigos necesarios tras la invocación al módulo de servicios.

4.2.3. Módulo middleware

A pesar de haber separado este módulo del módulo enrutador, este módulo está totalmente ligado a la utilización del protocolo HTTPS. Se trata de un módulo totalmente independiente del módulo enrutador, y tiene dos funciones principales:

- Interceptar todas las peticiones antes de que lleguen al enrutador y validar las cabeceras y el

token JWT. Si el token no es válido entonces se envía un 401 Unauthorized sin llegar al enrutador.

- Interceptar los errores que se provoquen durante la ejecución del programa (independientemente del módulo) y traducirlos a respuestas HTTPS. Para esto es necesario utilizar un modelo común de error que pueda ser interceptado por este módulo.

4.2.4. Módulo de servicios

En este módulo reside la totalidad de nuestra lógica de negocio. A este módulo ya llegan objetos modelados con nuestro modelo de datos, y es totalmente independiente del protocolo utilizado. Se encarga de hacer llamadas a los repositorios y de aplicar la lógica correspondiente en cada uno de los casos.

Un ejemplo de esta lógica es el registro de dispositivos; una vez recibido un dispositivo y sus correspondientes comandos, el servicio se encarga de hacer las comprobaciones correspondientes y guardar el dispositivo y después sus comandos.

Además, el módulo de servicios transforma los posibles errores provenientes de los repositorios para encapsularlos en errores internos. Un ejemplo sería transformar un error **"14 SQLITE_CANT_OPEN"** en el siguiente error: **"Error 01: no se ha podido acceder a la base de datos sqlite"**.

Tanto la entrada como la salida de datos de los métodos de nuestros servicios seguirán el modelo de datos del HUB.

4.2.5. Módulo repositorio

Este módulo contiene toda la gestión de los datos del HUB. Es invocado por el módulo de servicios, y es el encargado de gestionar las conexiones con la base de datos e insertar/obtener datos de la misma. Este módulo recibe datos modelados con nuestro modelo de datos, pero no necesariamente la manera de enviarlos/guardarlos tiene que coincidir con nuestro modelo de datos. Sin embargo, el retorno de los métodos de este módulo sí serán datos modelados.

Si en un futuro se realizasen llamadas a terceros, una API de Google por ejemplo, las llamadas a esas APIS también se realizarían desde este módulo.

4.2.6. Vista general

Por lo tanto, el diseño esquemático de la arquitectura interna del hub es el siguiente:

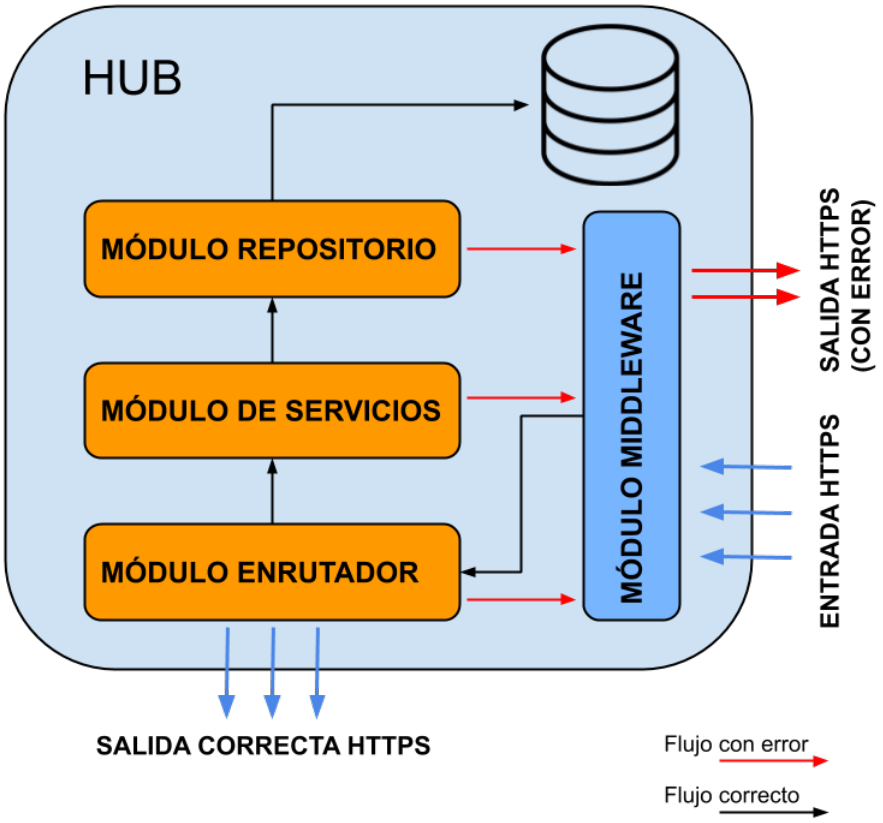


Figura 4.2: Arquitectura interna del hub

4.2.7. Seguridad

Para desarrollar una aplicación segura se ha optado por utilizar protocolo HTTPS y autenticación en cada petición mediante tokens JWT.

Los tokens JWT son tokens de acceso autocontenidos, es decir, contienen la información del usuario: identificador, rol...etc. Se utilizan algoritmos de cifrado para cifrar su contenido y evitar su modificación. Como ya se ha descrito, un módulo middleware de seguridad es el encargado de interceptar todas las peticiones y validar su token JWT. Para la validación y creación de los tokens JWT se ha utilizado la librería jsonwebtoken [9]. Se puede ver el uso de la librería en la Figura A.4.

Para el uso del protocolo HTTPS es necesario que el servidor cree un certificado de clave pública firmado por una entidad certificadora, de esta manera será aceptado por el navegador. En nuestro caso, debido a que somos un sistema interno, no necesitamos entidad certificadora y somos nosotros

mismos quienes firmamos el certificado. [1]

Para la adquisición del certificado autofirmado se ha utilizado OpenSSL [10] y llevado a cabo la siguiente acción: “openssl req -nodes -new -x509 -keyout server.key -out server.cert“. Este comando genera el certificado autofirmado y la clave privada.

Tras la generación del certificado, se ha reemplazado el antiguo módulo del servidor http por el nuevo módulo https, y se ha indicado la ruta del certificado y la clave privada. Este proceso se puede observar en la Figura A.5.

4.3. Interfaz

En esta sección se describirán los distintos componentes y pantallas que la aplicación tendrá, así como los diferentes servicios.

4.3.1. Pantallas

Como se ha explicado anteriormente, una de las grandes ventajas de Ionic es la enorme cantidad de UI Components ya hechos que nos ofrece. En esta sección, se explicarán las pantallas de la aplicación y el flujo entre ellas, y se explicarán los componentes utilizados para cada una de ellas.

Diagrama de flujo de las pantallas

Un diagrama de flujo nos ayuda a entender la navegabilidad de nuestra aplicación. Para evitar sobrecargar el diagrama se han omitido todas las pantallas que consisten en “pop-ups“ con diálogos de confirmación, selección, etc. a excepción de la pantalla de confirmación de eliminación del dispositivo (todas las pantallas de la aplicación pueden verse en el Apéndice B del documento).

Para la aplicación, se han diseñado cuatro pantallas principales:

- **Pantalla login**: esta pantalla nos permite acceder al sistema. Si los datos introducidos son correctos navegamos automáticamente a la pantalla de listado de dispositivos.
- **Pantalla listado de dispositivos**: es la pantalla principal de la aplicación. Nos permite visualizar todos los dispositivos y parte de su información (nombre, información, localización). Nos permite crear nuevas localizaciones, filtrar los dispositivos por localización y eliminar dispositivos. Se puede volver a la pantalla de login pulsando en el botón de logout, ir a la vista en detalle del

[1] Para poder realizar peticiones a un servidor HTTPS con un certificado autofirmado es necesario añadir excepciones en los navegadores y desactivar la opción de “validación de certificados“ de Postman.

dispositivo pulsando sobre un dispositivo, y navegar a la pantalla de configuración con el menú inferior de navegación.

- **Pantalla configuración**: comparte cabebcera y menú inferior de navegación con la pantalla listado de dispositivos. Nos permite, en el caso de que tengamos permisos, añadir usuarios, modificar nuestro usuario y reestablecer el HUB a su estado de fábrica.
- **Pantalla detalle dispositivo**: esta pantalla muestra en detalle la información de un dispositivo. Nos permite eliminarlo, editarlo y enviarle comandos. Además del nombre, la información y la localización podemos ver la hora última a la que el dispositivo envió información al HUB, así como su frecuencia de actualización.

Componentes

Para el desarrollo de la aplicación se han utilizado los siguientes UI Components de Ionic:

- **ion-button**: son un componente básico de la UI. Se trata de botones, pueden ser personalizados con iconos, colores, formas...etc.
- **ion-icon**: otro componente básico de la UI. Este componente nos permite encapsular iconos en componentes ionic. Ionic ofrece una librería de iconos bastante extensa, de la cual se han sacado todos los iconos de la aplicación: https://ionicons.com/.
- **ion-input**: son inputs encapsulados en componentes ionic. Existen diferentes tipos de inputs: text, password, email, etc. y pueden ser customizados de diferentes maneras. Se han utilizado, por ejemplo, en la pantalla de login, para introducir usuario y contraseña.
- **ion-tabs**: son un componente de navegación. Cada “tab“ corresponde a una pantalla, y navegar entre ellas a través de la barra inferior de navegación. En nuestro caso, la aplicación contiene dos tabs: “devices“ y “settings“, correspondientes a las pantallas listado de dispositivos y de configuración.
- **ion-header**: nos permite añadir una cabecera a la pantalla. En nuestro caso, la cabecera principal (igual para todas las tabs), contiene un botón de logout y el logo de la aplicación.
- **ion-cards**: se trata de un componente estándar a la hora de hacer interfaces en ionic, y sirven para agrupar información por bloques. En nuestro caso, se utilizan cards en las pantallas de listado de dispositivos y de información/edición del dispositivo. En la pantalla de listado de dispositivos cada dispositivo corresponde a una “ion-card“. Tienen subcomponentes como: header, title, subtitle, content, etc.
- **ion-list**: este componente nos permite hacer listas de otros componentes. En el caso de nuestra aplicación lo usamos en la pantalla de listado de dispositivos, donde hacemos una lista de “ion-cards“; y en la pantalla de configuración, donde hacemos una lista de “ion-buttons“.
- **ion-item-sliding**: son componentes deslizables. Se puede customizar de diferentes maneras, y añadir callbacks o componentes ocultos que aparecen cuando el componente padre se desliza. En nuestro caso, en la pantalla de dispositivos, cada uno de ellos, a parte de ser un ion-card es un item-sliding, lo que nos permite deslizar el dispositivo a la izquierda para que se muestre el botón de eliminar dispositivo.

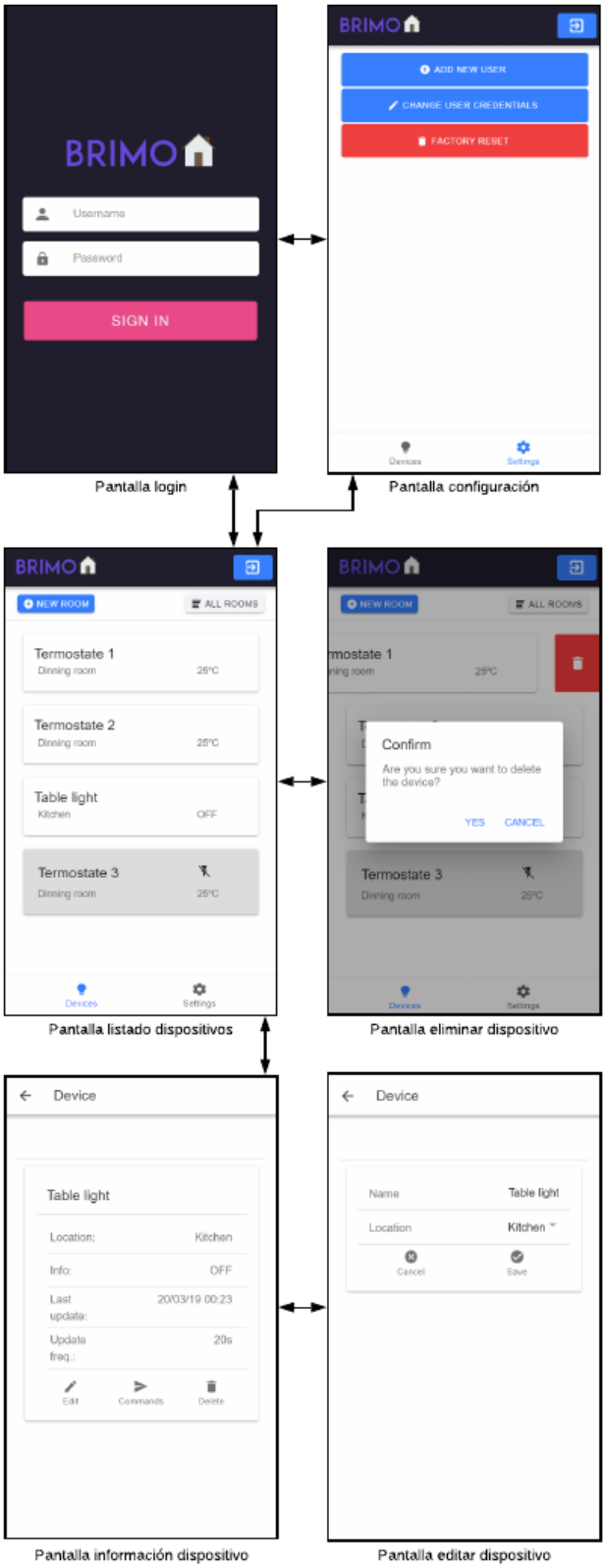


Figura 4.3: Flujo de pantallas de la interfaz

En el siguiente ejemplo se muestran los componentes utilizados en la pantalla de listado de dispositivos:

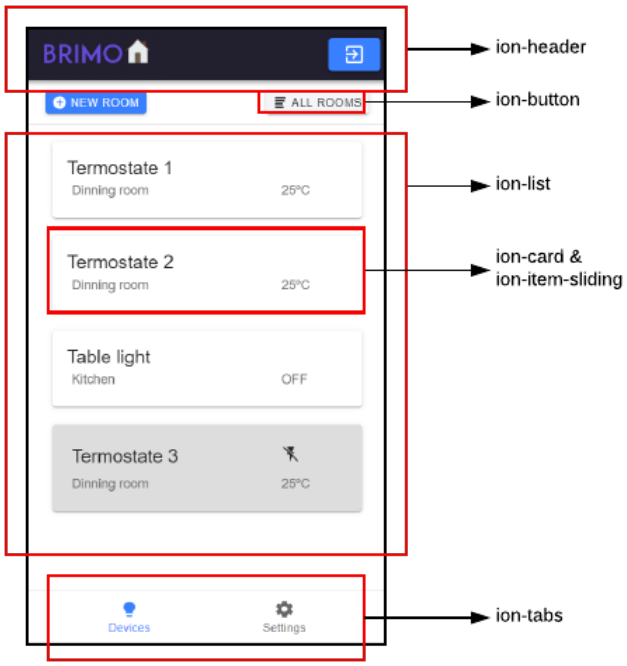


Figura 4.4: Componentes utilizados en la pantalla listado de dispositivos

4.3.2. Servicios

Los servicios sirven para compartir funcionalidad entre los componentes. En nuestro caso, serán los encargados de comunicarse con el exterior, es decir, de hacer peticiones al HUB. Los servicios son independientes de la interfaz, es la interfaz quien los invoca. Por ejemplo, en la pantalla de login, la interfaz se encarga de recoger los datos (usuario y contraseña) e invocar al servicio de login, que tras hacer una petición al HUB devolverá una respuesta que será procesada de nuevo por la interfaz, mostrando un mensaje de error en caso de fallo, o navegando a la pantalla principal en caso de éxito.

Para el desarrollo de nuestra aplicación hemos creado dos servicios que se encargarán de comunicarse con el HUB y un servicio para comunicar componentes entre sí:

- **Servicio de autenticación**: se encarga de comprobar si el usuario está logado y de recuperar el token JWT. Además, este servicio hace las llamadas de login y registrar/modificar usuario. Cuando se hace una llamada de login, el token JWT obtenido se guarda en la memoria del dispositivo, para ser recuperado cada vez que se hace una petición.
- **Servicio de comunicación**: se encarga de comunicarse directamente con el HUB. Utiliza el servicio de autenticación para recuperar el token JWT y añadirlo en las cabeceras de cada petición.

Este servicio sirve para añadir habitaciones, eliminar dispositivos, editar dispositivos...etc.

- **Servicio compartido**: este servicio se encarga de comunicar componentes entre sí. Funciona con observables, a los que cada componente se suscribe en el caso de ser necesario. Funciona, por ejemplo, para saber que el token ha caducado: si cualquier petición recibe un 401 entonces envía un evento a este servicio para que el resto de componentes (que se haya suscrito a dicho tipo de eventos) lo sepa y la aplicación redirija automáticamente a la pantalla de login.

5 Pruebas

5.1. Introducción

En este capítulo se describen las pruebas realizadas para la verificación del correcto funcionamiento del sistema.

Para verificar el correcto funcionamiento del sistema se han realizado tests durante todo el desarrollo. Antes de comenzar a describir las pruebas realizadas es necesario diferenciar entre diferentes tipos de tests:

- Tests unitarios: sólo prueban una clase/método/funcionalidad del sistema. El resto de componentes del sistema se maquetan para simular su comportamiento. Este tipo de tests verifica el correcto comportamiento de una parte del sistema, no del sistema completo. Son muy interesantes para testear métodos complejos con mucha casuística.
- Tests de integración: prueban la totalidad del sistema. A diferencia de los tests unitarios, no se maquetan elementos internos del sistema, se prueban todos los elementos juntos.
- Tests de interfaz: prueban que la visualización de la interfaz sea correcta en diferentes dispositivos y resoluciones.

En el Apéndice C de este documento se encuentran ejemplos de las pruebas realizadas.

5.2. Pruebas unitarias

Durante la realización de la API se han creado tests unitarios que prueban cada método de manera independiente, sin tener en cuenta el resto de componentes del sistema. Para ello, hemos utilizado la librería Chai [11], que nos ayuda con el testing de la aplicación.

Se han testeado métodos de entrada de la API y verificado las respuestas y errores que se debían devolver.

Durante estas pruebas se han detectado respuestas y fallos no esperados. Por ejemplo, un error **500 INTERNAL SERVER ERROR** al no encontrar un dispositivo, en lugar de **404 NOT FOUND**.

5.3. Pruebas de integración

Para los tests de integración de la API se ha creado una colección de Postman de manera incremental, donde se prueba cada uno de los métodos de la API que se han ido desarrollando. Además, para comprobar el correcto funcionamiento de los accesos a base de datos se ha utilizado “DB Browser for SQLite“.

Una vez probado cada método individualmente se pasa el Collection Runner de Postman [12], que ejecuta todos los métodos que existen en una colección. Todos los métodos de la colección llevan un test que verifica que el código de respuesta sea correcto, y se pueden añadir tests específicos para ese método.

En este caso se ha creado un flujo que lleva a cabo las siguientes acciones:

1. Login: se hace login con el usuario por defecto y se guarda el token generado en la variable “ACCESS_TOKEN“ para usarse en las peticiones posteriores.
2. Nuevo usuario: se añade un usuario nuevo utilizando el token anteriormente generado. El id del usuario creado se guarda en la variable “USER_ID“.
3. Modificación usuario: se modifica el usuario creado, accediendo a él a través de su identificador previamente guardado.
4. Eliminación usuario: se elimina el usuario creado, accediendo a él a través de su identificador previamente guardado.
5. Añadir dispositivo: se añade un nuevo dispositivo y se guarda su identificador en la variable “DEVICE_ID“.
6. Actualizar info dispositivo: se actualiza la información del dispositivo utilizando su identificador. Se envía como información la palabra “updatedinfo“.
7. Listar dispositivos: se obtiene el listado de dispositivos. El primer dispositivo de la lista debe tener la palabra “updatedinfo“ en el campo información.
8. Obtener información dispositivo: se obtiene la información del dispositivo a través de su identificador. Esta información debe contener la información actualizada.
9. Añadir localización nueva: se añade una nueva localización y se salva su identificador en la variable “ROOM_ID“.

10. Editar localización: se edita la localización añadida a través de su identificador, y se cambia su nombre a “new_room_updated“.
11. Listar localizaciones: se obtiene el listado de localizaciones, que debe contener la localización añadida y editada anteriormente.
12. Editar dispositivo: se edita la localización del dispositivo añadido con la nueva localización.
13. Eliminar dispositivo: se elimina el dispositivo.
14. Eliminar localización: se elimina la localización añadida anteriormente.
15. Enviar comando a dispositivo: se envía un comando al dispositivo, este test debe fallar debido a que no existe ningún dispositivo real que recepcione el comando.

Antes de pasar los tests de integración se debe eliminar la base de datos, y al finalizar la ejecución de los tests la base de datos debe quedar vacía, a excepción del usuario que se crea por defecto.

5.4. Pruebas de interfaz

Estos tests se han pasado gracias a la herramienta Vista de Diseño Adaptable [13] de Firefox, que emula la visualización de la interfaz en distintos dispositivos móviles, ya que no se dispone de recursos suficientes para adquirir dispositivos físicos.

Durante la realización de estos tests se ha probado la aplicación en dispositivos móviles y en tablets, con sistemas operativos Android e IOS. Este se debe a que los componentes Ionic tienen una estética diferente dependiendo del sistema operativo.

Durante las pruebas se han detectado errores en la adaptación de la interfaz a diferentes dispositivos: botón de login descentrado, botones sin mostrar los iconos...etc. La mayoría de estos errores se han solventado modificando el CSS de la aplicación, aunque en algunos casos ha sido necesario añadir componentes internos para la correcta visualización de la interfaz.

6

Trabajos futuros y conclusiones

6.1. Trabajos futuros

La domótica es un campo muy amplio y en constante cambio. Debido a la amplitud del campo y a su joven vida, BRIMO tiene todavía un largo recorrido en cuanto a mejoras y nuevas funcionalidades se refiere. En esta sección se hablará de manera breve sobre algunas de las posibles mejoras y trabajos futuros.

6.1.1. Mejoras

Durante el desarrollo de BRIMO, y especialmente en la etapa de integración con la interfaz, me he dado cuenta de que muchas veces tenía dificultades para trazar los errores. Por un lado, si se producía un error en el HUB era necesario mirar la consola o la respuesta que daba la llamada en cuestión, y si el error se producía en el dispositivo había que mirar los logs de los dispositivos directamente.

Para lidiar con este problema, una solución práctica sería añadir un nuevo microservicio para el registro de errores. Este nuevo microservicio, que se ejecutaría en paralelo al HUB, tendría su propia base de datos de errores, utilizando un modelo común de log (timestamp, mensaje y módulo de error), y sería capaz de:

- Guardar errores externos: los errores producidos en el HUB o en los dispositivos se mandarían vía endpoint a este nuevo microservicio, que los almacenaría en su base de datos.

- Servir los errores almacenados: la interfaz debería ser capaz de acceder a estos logs y paginarlos, buscar por fecha, por módulo de error...etc.

De esta manera, todos los logs estarían unificados y serían accesibles desde la interfaz.

6.1.2. Trabajos futuros

En cuanto a trabajos futuros, existen muchas funcionalidades que se pueden añadir al HUB para mejorar la experiencia de usuario. Algunos de los posibles trabajos futuros son:

- Control por voz: añadir al HUB control por voz sería una funcionalidad muy cómoda para el usuario. Sería necesario añadir un micrófono a la placa, ya que la Raspberry no lo incorpora por defecto. Además, habría que integrar un nuevo módulo de control por voz que accedería al módulo de servicios tal y como lo hace actualmente el módulo enrutador.
- Módulo ZigBee: actualmente el HUB sólo acepta dispositivos que envíen y reciban información vía HTTPS. Sin embargo, añadir un nuevo módulo para dispositivos con tecnología ZigBee permitiría al HUB ampliar su rango de dispositivos de manera muy notable. Para esta integración sería necesario adquirir un nuevo componente llamado “Raspbee“ [14] para que la Raspberry pudiera comunicarse con dispositivos con tecnología ZigBee, y hacer la codificación necesaria para poder integrarlo con el resto del HUB.
- Cámaras IP: actualmente el HUB no permite integrar cámaras IP. Se trata de un dispositivo domótico muy común que transmite mucha seguridad al usuario. Permitir cámaras IP supondría un gran avance para el HUB.
- Programación de comandos: permitir al usuario programar el envío de comandos a sus dispositivos es una funcionalidad realmente atractiva para el usuario, ya que facilitia al usuario la gestión de su hogar. Ejemplos de programación de comandos serían: “encender las luces de lunes a viernes a las 07:30“, “encender el termostato dentro de una hora“...etc. Esta programación de tareas sería posible gracias al módulo NPM node-cron [15]

6.2. Conclusiones

Durante el desarrollo de este trabajo se ha desarrollado un sistema para la gestión de dispositivos domóticos. Este sistema funciona como intermediario entre dispositivos y usuario, y tiene dos partes principales: el HUB y la interfaz.

En el HUB se ejecuta el core de la aplicación, se realiza la comunicación con los dispositivos y se guarda la información de los mismos. La interfaz es la encargada de la comunicación entre el HUB y el usuario. Todas las comunicaciones se realizan mediante peticiones HTTPS.

El desarrollo de este proyecto me ha permitido mejorar las habilidades de análisis y diseño de software, ya que una gran parte del proyecto ha consistido en analizar y diseñar el sistema.

Además, he descubierto y aprendido tecnologías actuales y muy utilizadas actualmente, como son Node.js e Ionic.

Por último, durante el estudio del estado del arte, he adquirido conocimientos sobre domótica que antes no conocía. El desarrollo de este proyecto me ha generado curiosidad e interés por la domótica, lo que me ha permitido trabajar de manera eficaz y, sobre todo, con entusiasmo.

No cabe duda de que la domótica está a la orden del día, pero todavía tiene un larguísimo recorrido por delante. Estoy convencido de que en un futuro próximo, la domótica formará parte de nuestros hogares y de nuestras ciudades, ayudándonos a mantener ciudades y hogares cómodos, eficientes, seguros y accesibles para todos.

Glosario de acrónimos

- **Sistema domótico**: Un sistema domótico es el conjunto de controladores y dispositvios que hacen posible la automatización del hogar.
- **Dispositivo domótico**: Dispositivo que nos ayuda a la automatización del hogar actuando o recopilando información. Ejemplos: sensores de temperatura, relés, cámaras...etc.
- **Bridge**: Dispositivo que nos ayuda a controlar y administar diferentes dispositivos domóticos. Los dispositivos se conectan al bridge, y el cliente interactua directamente a través de él.
- **REST**: Arquitectura software que se apoya en el protocolo HTTP. Se utiliza en arquitecturas cliente-servidor. El cliente tiene operaciones básicas y predefinidas: GET, POST, PUT, DELETE... Y el sevidor responde a las peticiones con su correspondiente código HTTP. Cada recurso del servidor es direccionable a través de su URI.
- **JSON**: JavaScript Object Notation: formato de texto sencillo para el intercambio de datos.
- **API**: Del inglés Application Programming Interface: conjunto de funciones y procedimientos que pueden ser utilizadas por otro software.
- **SPA**: Del inglés Single Page Application: aplicación web que se ejecuta en una sola página, sin necesidad de refrescar el navegador, haciendo más fluida la navegación.
- **MVC**: Modelo Vista Controlador: arquitectura software que separa los datos (Modelo) de la interfaz de usuario (Vista), su comunicación y lógica se encuentra en el controlador.
- **Responsive**: Diseño web cuyo objetivo es adaptar la apariencia de la página web a diferentes dispositivos.
- **Man-In-The-Middle**: Tipo de ataque en el que el atacante es capaz de interceptar y/o modificar mensajes enviados entre dos partes. Comúnmente este ataque se realiza en redes donde se utilizan protocolos HTTP, el atacante es capaz de captar las peticiones y modificarlas.

Bibliografía

[1] C. R. Española, "¿Qué es la teleasistencia?." https://www.cruzroja.es/principal/web/teleasistencia/que-es-la-teleasistencia, 2018. [Online; accessed 15-February-2019].

[2] I. StrongLoop, "Documentación Express." https://expressjs.com/es/, 2019. [Online; accessed 25-March-2019].

[3] A. Goel, "Top 10 Web Development Frameworks in 2019." https://hackr.io/blog/top-10-web-development-frameworks-in-2019, 2019. [Online; accessed 23-March-2019].

[4] I. NPM, "About npm." https://docs.npmjs.com/about-npm/, 2019. [Online; accessed 25-March-2019].

[5] I. Stack Exchange, "Developer Surver Results 2018.." https://insights.stackoverflow.com/survey/2018#technology, 2018. [Online; accessed 23-March-2019].

[6] S. org3, "About SQLite." https://www.sqlite.org/about.html, 2019. [Online; accessed 23-March-2019].

[7] S. org3, "SQLite3 NPM Module documentation.." https://www.npmjs.com/package/sqlite3, 2019. [Online; accessed 23-March-2019].

[8] I. Ionic, "What is Ionic Framework?." https://ionicframework.com/docs/intro, 2019. [Online; accessed 23-March-2019].

[9] "JsonWebToken NPM Module documentation." https://www.npmjs.com/package/jsonwebtoken, 2019. [Online; accessed 18-March-2019].

[10] O. S. Foundation, "OpenSSL documentation." https://www.openssl.org/docs/, 2019. [Online; accessed 18-March-2019].

[11] "Chai Assertion Library Documentation.." https://www.chaijs.com/api, 2019. [Online; accessed 30-March-2019].

[12] I. Postman, "Documentación Collection Runs." https://learning.getpostman.com/docs/postman/collection_runs/intro_to_collection_runs/, 2019. [Online; accessed 30-March-2019].

[13] "Responsive Design View." https://developer.mozilla.org/es/docs/Tools/Responsive_Design_View, 2019. [Online; accessed 30-March-2019].

[14] D. E. GMBH, "The Raspberry Pi Zigbee gateway ." https://phoscon.de/en/raspbee, 2019. [Online; accessed 20-March-2019].

[15] L. Merencia, "NodeCron NPM Module documentation." https://www.npmjs.com/package/node-cron, 2018. [Online; accessed 30-March-2019].

[16] Wikipedia, "Domótica." https://es.wikipedia.org/wiki/Domótica, 2019. [Online; accessed 10-February-2019].

[17] O. Pablo Domínguez, "En qué consiste el modelo en cascada." https://openclassrooms.com/en/courses/4309151-gestiona-tu-proyecto-de-desarrollo/4538221-en-que-consiste-el-modelo-en-cascada, 2017. [Online; accessed 3-March-2019].

[18] E. e. S. D. HogarTec, "Sistemas domóticos centralizados, descentralizados y distribuidos." https://hogartec.es/hogartec2/sistemas-domoticos-centralizados-descentralizados-y-distribuidos/, 2019. [Online; accessed 3-March-2019].

[19] M. Coppock, "Understanding Smart Home Communication Protocols." https://www.newegg.com/insider/understanding-smart-home-communication-protocols/, 2017. [Online; accessed 3-March-2019].

[20] D. Integrada, "Instalaciones domóticas: Tipos y elementos que debes conocer." https://domoticaintegrada.com/instalaciones-domoticas/, 2017. [Online; accessed 7-March-2019].

[21] E. a. I. Manuel J. Gutiérrez, "Todo sobre ZigBee, la tecnología ultrabarata para comunicación inalámbrica." https://elandroidelibre.elespanol.com/2015/08/todo-sobre-zigbee-la-tecnologia-ultrabarata-para-comunicacion-inalambrica.html, 2015. [Online; accessed 10-March-2019].

[22] Wikipedia, "Bluetooth de baja energía." https://es.wikipedia.org/wiki/Bluetooth_de_baja_energía, 2019. [Online; accessed 10-March-2019].

[23] I. SmartLabs, "Insteon: the technology." https://www.insteon.com/technology, 2019. [Online; accessed 10-March-2019].

[24] I. Electronic Design, Informa USA, "INSTEON Technology." https://www.electronicdesign.com/embedded/refresh-insteon-technology, 2006. [Online; accessed 10-March-2019].

[25] Z-Wave, "Z-Wave: Learn." https://www.z-wave.com/learn, 2019. [Online; accessed 12-March-2019].

[26] R. Vega, "Z-Wave: un protocolo inalámbrico para la domótica." https://ricveal.com/blog/z-wave/, 2014. [Online; accessed 12-March-2019].

[27] F. Copes, "How to create a self-signed HTTPS certificate for Node.js to test apps locally." https://flaviocopes.com/express-https-self-signed-certificate/, 2018. [Online; accessed 18-March-2019].

[28] S. Afolaranmi, "Testing A Node/Express Application With Mocha & Chai." https://scotch.io/tutorials/how-to-test-nodejs-apps-using-mocha-chai-and-sinonjs, 2018. [Online; accessed 30-March-2019].

Anexos

A

Código desarrollo

A.1. Código API

```
var express = require('express');
var router = express.Router();
var locationsService = require('../services/locations.service');


/**
 * Route GET /locations. Get locations list.
 * Return 200 if success.
 */
router.get('/', function (req, res, next) {
  locationsService.list().then(response => {
    if (response.length == 0) {
      res.status(204);
      res.json();
    } else {
      res.json(response);
    }
  }).catch(err => next(err));
});

/**
 * Route POST /locations. Create location.
 * Return 201 if success.
 */
router.post('/', function (req, res, next) {
  let location = req.body;
  if (!location.descr) return next({status: 400, message:'Missing descr.'});
  locationsService.create( location ).then(response => {
    res.status(201);
    res.json(response);
  }).catch(err => next(err));
});

/**
 * Route PUT /locations/{id}. Edit location information.
 * Return 201 if success.
 */
router.put('/:id', function (req, res, next) {
    let locationInfo = req.body;
    if (!locationInfo.descr) return next({status: 400, message:'Missing descr.'});
        locationsService.editLocation( locationInfo, req.params.id ).then(response => {
        res.status = 201;
        res.json(response);
    }).catch(err => next(err));
});

/**
 * Route DELETE /locations/{id}. Delete location.
 * Return 201 if success.
 */
```

Figura A.1: Ejemplo módulo enrutador: locations routes

```
/**
 * Creates a device into database. If commands field is set, then inserts each commands on commands database.
 * Throws 500 error if error.
 * @param {Object} device
 */
function create(device) {
  var room = null;
  if (device.room != null && device.room.id != null) {
    console.log(device.room.id);
    return locationsRep.findById(device.room.id).then(room => {
      return devicesRep.create(device, room).then(dev => {
        if (device.commands != null) {
          device.commands.forEach(command => commandsRep.insertCommand(dev.device_id, command).catch(err => console.log(err)));
        }
        return dev;
      });
    });
  }
  return devicesRep.create(device, null).then(dev => {
    if (device.commands != null) {
      device.commands.forEach(command => commandsRep.insertCommand(dev.device_id, command).catch(err => console.log(err)));
    }
    return dev;
  });
}

/**
 * Edit device.info field and update lastupdate field.
 * @param {Object} device
 * @param {int} id
 */
function editInfo(info, deviceId) {
  return devicesRep.editInfo(info, deviceId).catch(err => {
    throw {
      status: 400,
      message: err
    };
  });
}
```

Figura A.2: Ejemplo módulo servicios: devices service

```
const sqlite3 = require('sqlite3').verbose();
var db = null;

/**
 *  Creates DB connection. Throws error if connections fails.
 * @param {*} database the database from settings.
 */
function connect(database) {
  db = database;
  db.run("CREATE TABLE IF NOT EXISTS commands (command_id INTEGER PRIMARY KEY AUTOINCREMENT, command_descr text NOT NULL, command_code TEXT NOT NULL, device
    if (err) {
      console.error(err.message);
      throw err;
    }
  });
}

/**
 * Insert a command into commands table.
 * @param {*} device_id the device that command refereneces.
 * @param {*} command the command object.
 */
function insertCommand(device_id, command) {
  if ( device_id == null || command == null || command.command_code == null || command.command_descr == null){
    throw {
      status: 403,
      message: "Error inserting command"
    };
  }
  return new Promise((resolve, reject)=>{
    db.run("INSERT INTO commands (device_id, command_descr, command_code) VALUES (?1,?2,?3)",
    {
      1: device_id,
      2: command.command_descr,
      3: command.command_code
    }, function (err, rows) {
      if (err) {
        reject(err);
      }
      resolve({ command_id: this.lastID});
    });
  });
}
```

Figura A.3: Ejemplo módulo repositorio: devices repository

auth.token.js

```
var jwt = require('jsonwebtoken');
var createError = require('http-errors');
var config = require('./config');
const hoursToExpire = 1;

function createToken(user) {
  console.log('creating token with data:');
  console.log(user);
  return jwt.sign({
    user
  }, config.TOKEN_SECRET, {
    expiresIn: 60 * 60 * hoursToExpire
  });
};

function protected(req, res, next) {
  var token = req.headers['x-access-token'];
  if (!token) return next({status: 401, message: 'Unauthorized'});
  jwt.verify(token, config.TOKEN_SECRET, function (err, decoded) {
    if (err) return next({status: 401, message: 'Unauthorized'});
    req.verified = true;
    req.token = decoded;
    next();
  });
}

```

Figura A.4: Middleware autenticación

```
var app = require('../app');
var debug = require('debug')('refactorserver:server');
var https = require('https');
var fs = require('fs');

/**
 * Get port from environment and store in Express.
 */

var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);

//OLD HTTP SERVER: var server = http.createServer(app);

var server = https.createServer({
  key: fs.readFileSync('./certs/server.key'),
  cert: fs.readFileSync('./certs/server.cert')
}, app);
```

Figura A.5: Configuración HTTPS

A.2. Código Interface

```
export class AuthenticationService {
  url: string;

  constructor (public http: HttpClient) {
    this.url = URLAPI;
  }
  login(auth_object) {
    const url_login = this.url + '/login';
    return this.http.post(url_login, auth_object).toPromise().then( (response: any) => {
      localStorage.setItem(token_key, response.tkn_auth);
      return true;
    }).catch( () => false);
  }
  logout() {
    localStorage.removeItem(token_key);
  }
  isLogged() {
    const urlget = this.url + '/devices';
    const headers = this.getHeaders();
    if (!headers) { throw -1; }
    return this.http.get(urlget, headers).toPromise().then( (response: any) => {
      if (response.status === 401) {
        return false;
      }
      return true;
    }).catch(() => false );
  }
  getToken() {
    return localStorage.getItem(token_key);
  }
  getHeaders() {
    const token = this.getToken();
    if (!token) { return undefined; }
    const httpOptions = {
      headers: new HttpHeaders({
        'Content-Type':  'application/json',
        'x-access-token': token,
      })
    };
    return httpOptions;
  }
}
```

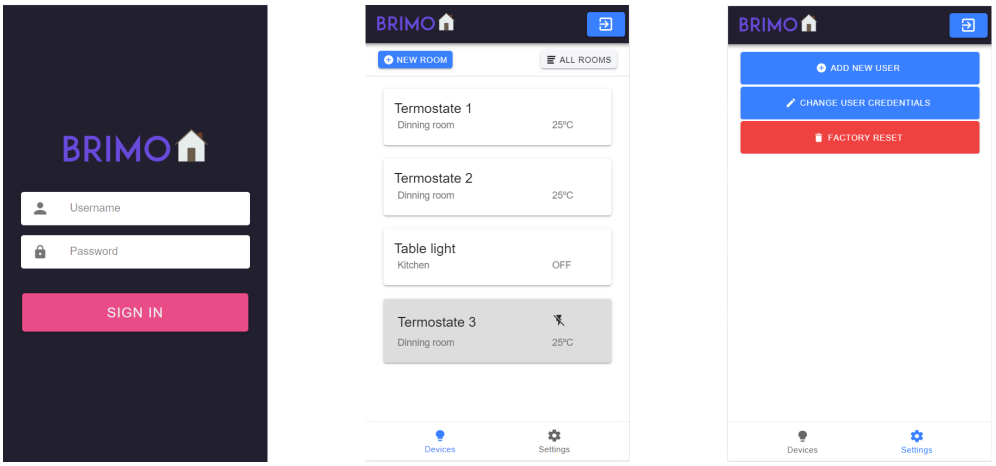
Figura A.6: Ejemplo servicio interfaz: servicio autenticación

```
<ion-tabs>

  <ion-tab-bar slot="bottom">
    <ion-tab-button tab="tab1">
      <ion-icon name="bulb"></ion-icon>
      <ion-label>Devices</ion-label>
    </ion-tab-button>

    <ion-tab-button tab="tab3">
      <ion-icon name="settings"></ion-icon>
      <ion-label>Settings</ion-label>
    </ion-tab-button>
  </ion-tab-bar>

</ion-tabs>
```

Figura A.7: Ejemplo componentes Ionic: tabs ionic

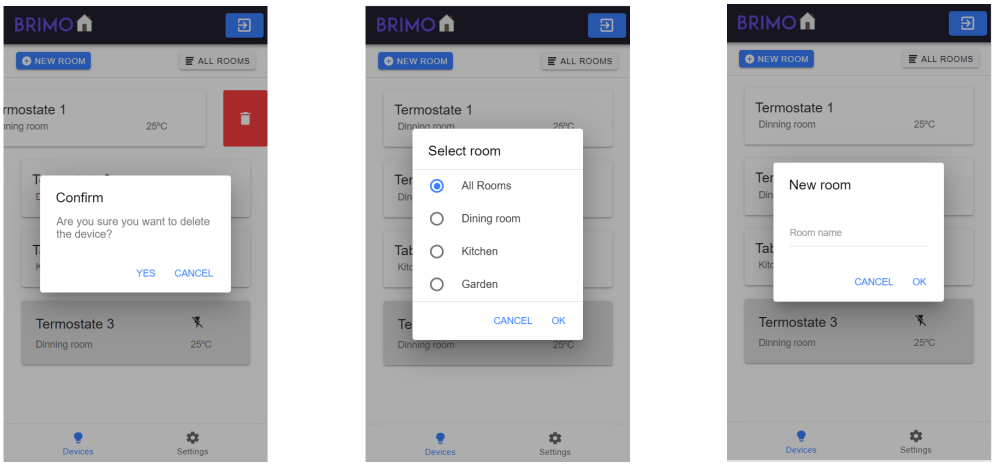
B

Pantallas aplicación



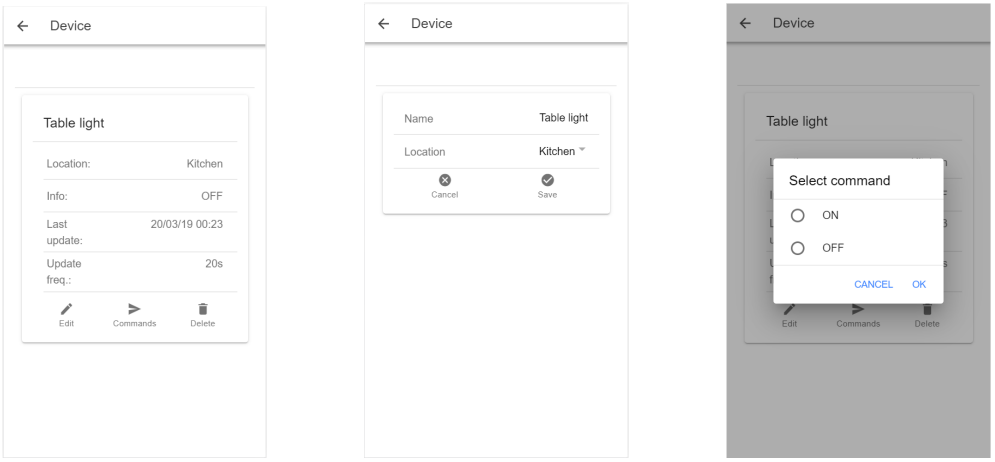
(a) Pantalla de login a la aplicación (b) Pantalla listado de dispositivo (c) Pantalla de ajustes del HUB

Figura B.1: Pantallas login, listado dispositivos y ajustes



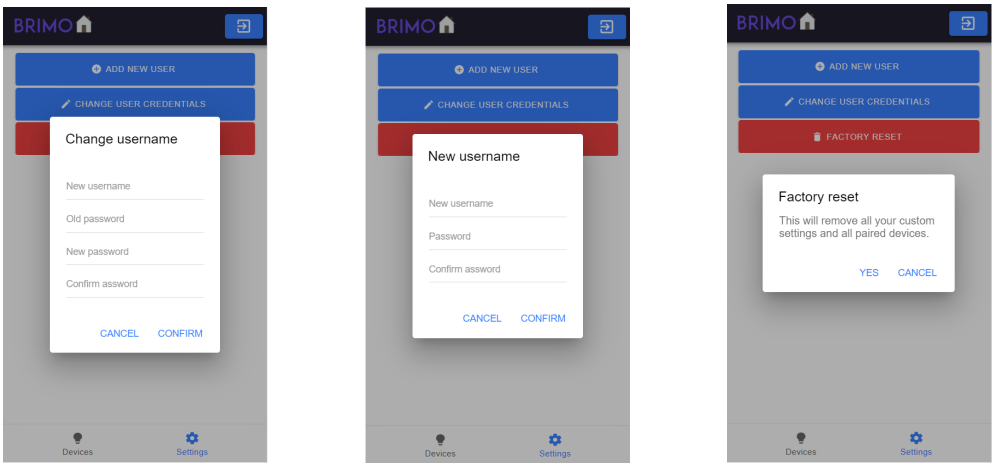
(a) Vista confirmar eliminar dispositivo (b) Pantalla filtrar por habitación (c) Pantalla crear habitación

Figura B.2: Pantallas eliminar dispositivo, filtrar por habitación y crear habitación



(a) Vista individual dispositivo (b) Pantalla editar dispositivo (c) Pantalla enviar comando

Figura B.3: Pantallas vista dispositivo, editar dispositivo y enviar comando



(a) Diálogo editar usuario (b) Pantalla añadir usuario (c) Pantalla resetear HUB

Figura B.4: Pantallas editar usuario, añadir usuario y resetear HUB

C

Pruebas realizadas

C.1. Pruebas unitarias

C.2. Pruebas integración con Postman

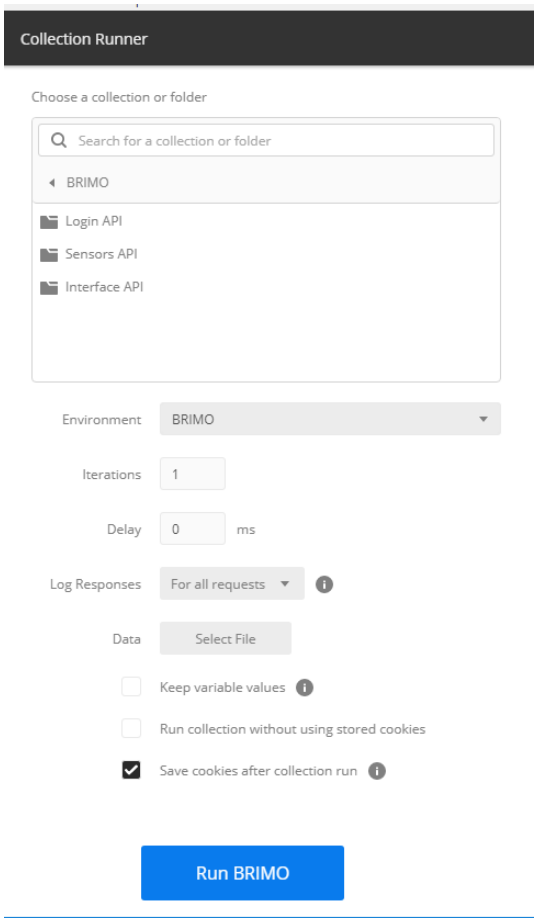


Figura C.1: Configuración collection runner

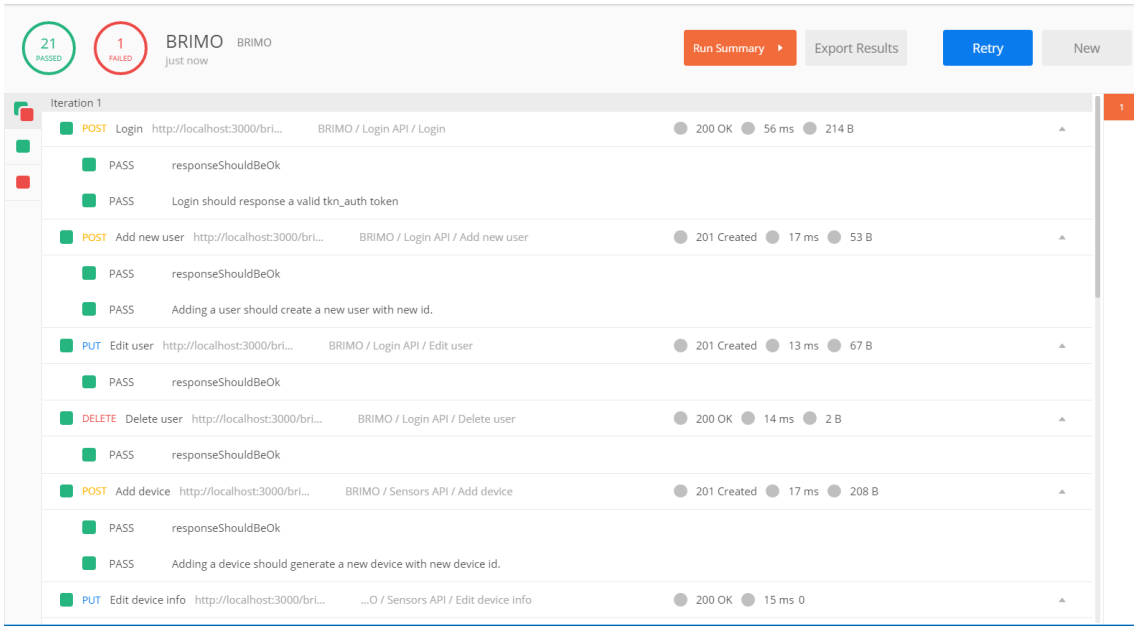
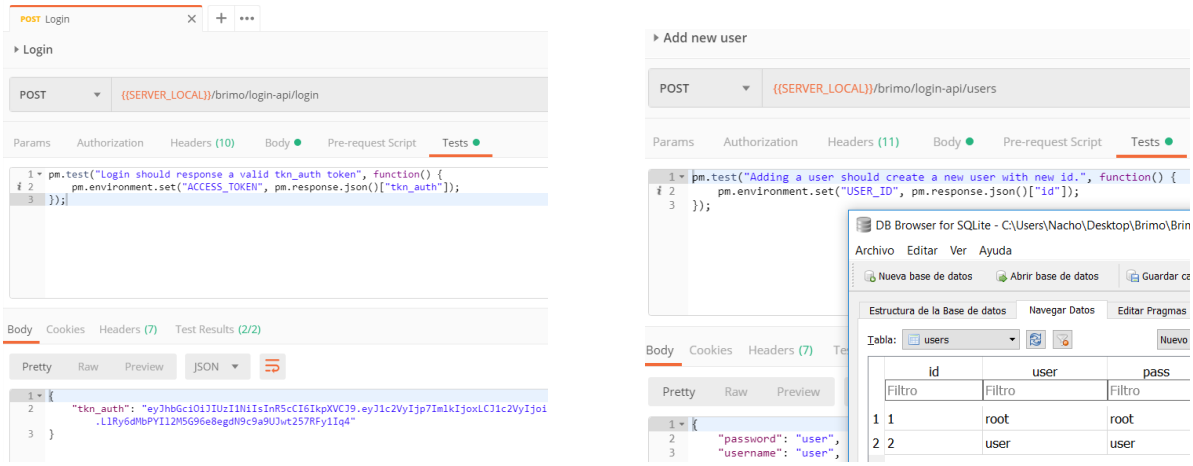


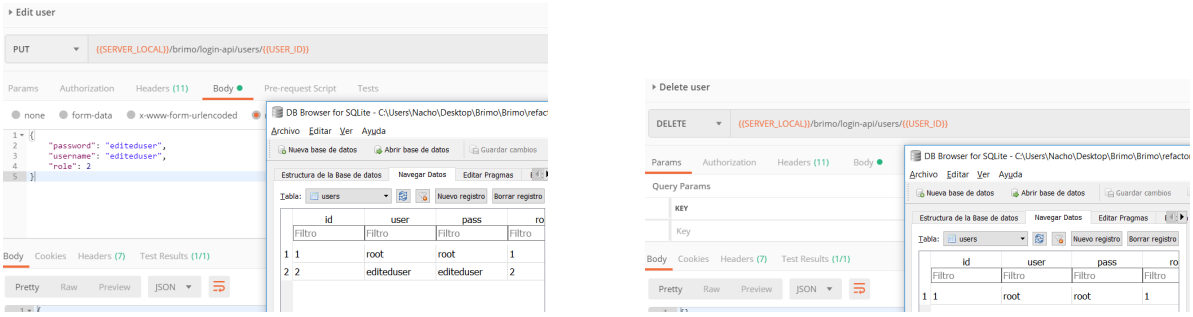
Figura C.2: Resultados collection runner



(a) Test login aplicación.

(b) Test añadir usuario.

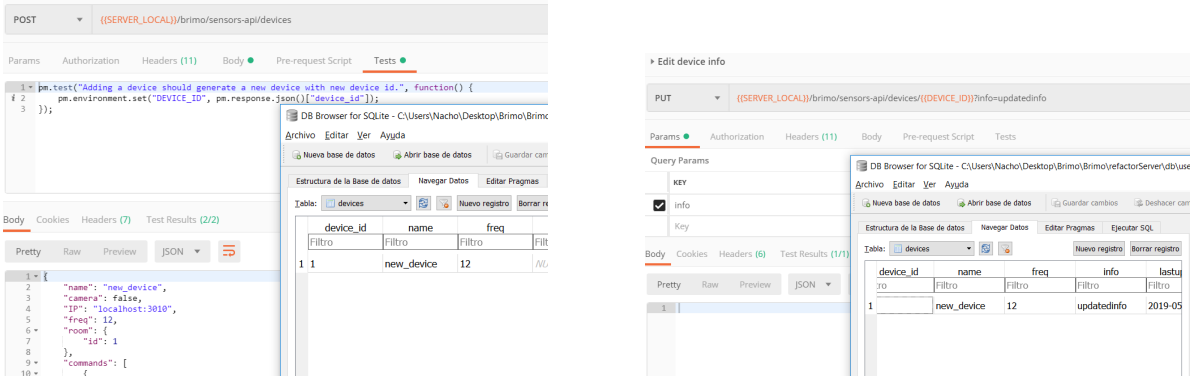
Figura C.3: Tests login y añadir usuario



(a) Test editar usuario

(b) Test eliminar usuario

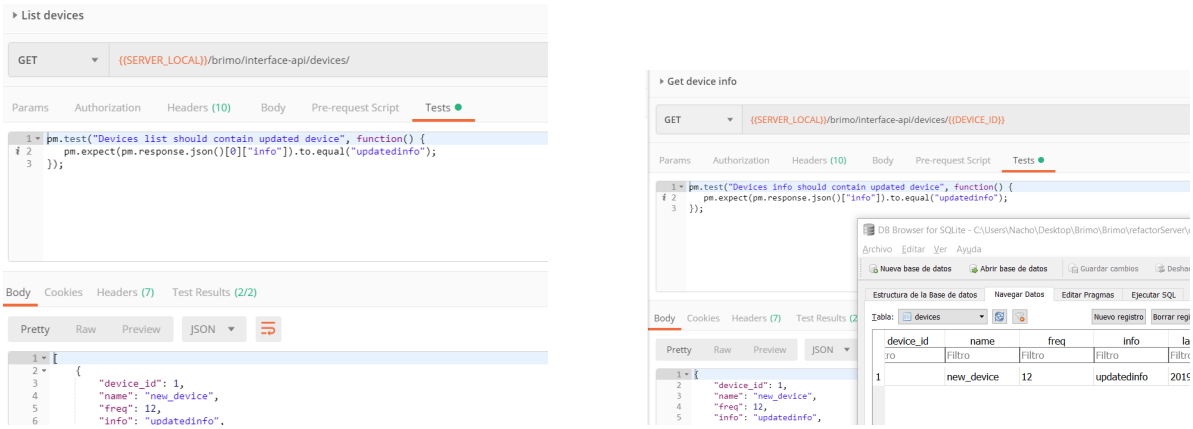
Figura C.4: Tests editar y eliminar usuario



(a) Test añadir dispositivo

(b) Test editar dispositivo

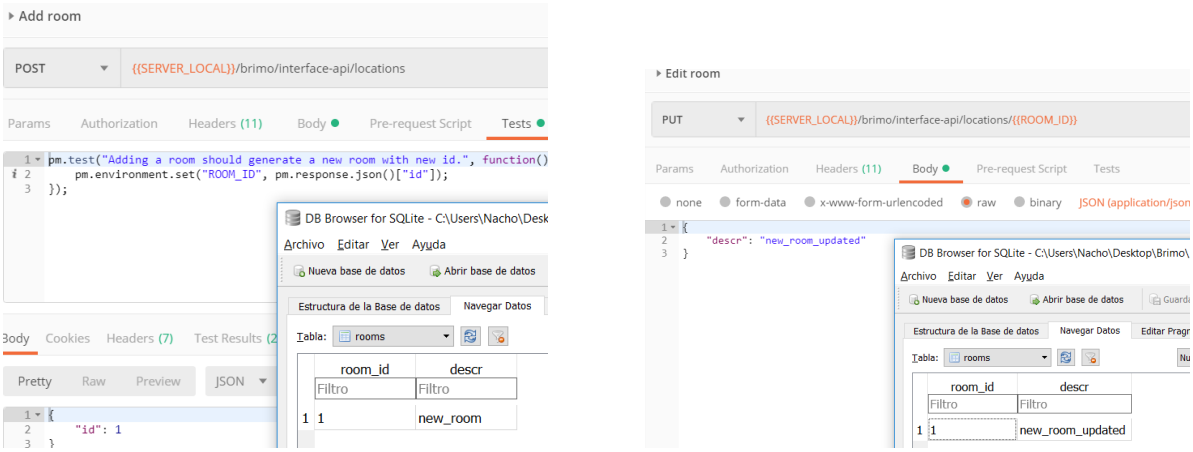
Figura C.5: Tests añadir y editar dispositivo



(a) Test listar dispositivos

(b) Test información dispositivo

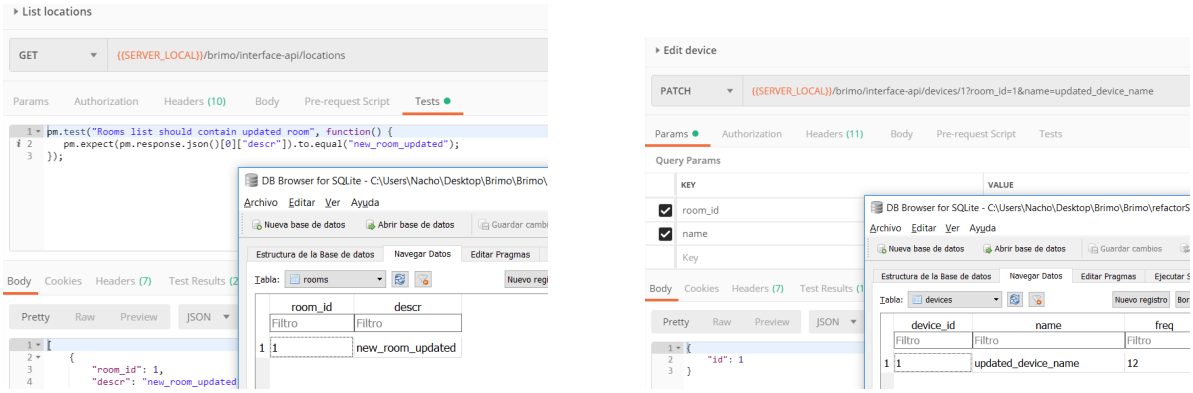
Figura C.6: Tests listar dispositivos y obtener información dispositivo



(a) Test añadir localización

(b) Test editar localización

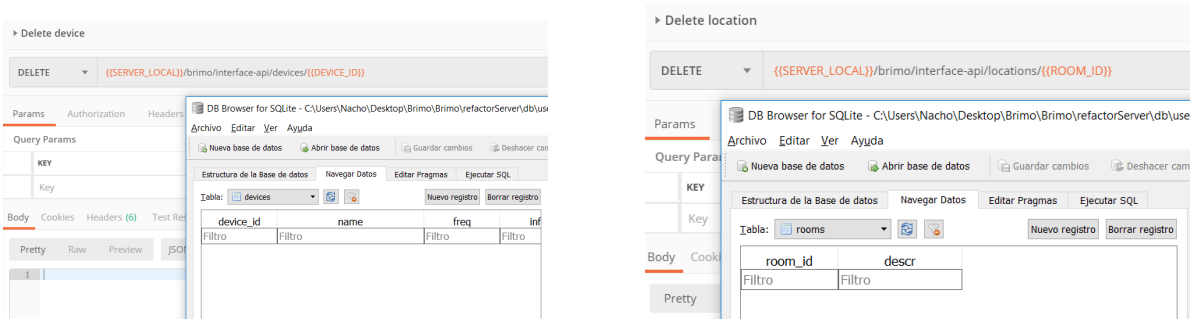
Figura C.7: Tests añadir y editar localización



(a) Test listar localizaciones

(b) Test editar nombre dispositivo

Figura C.8: Tests listar localizaciones y editar nombre dispositivo

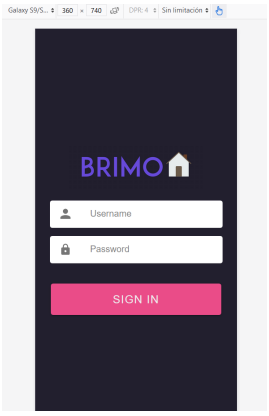


(a) Test eliminar dispositivo

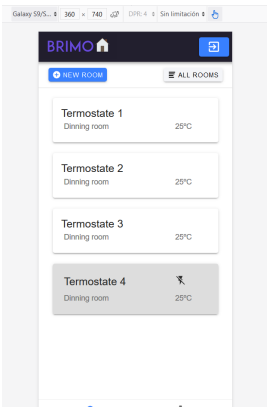
(b) Test eliminar localización

Figura C.9: Test eliminar dispositivo y eliminar localización

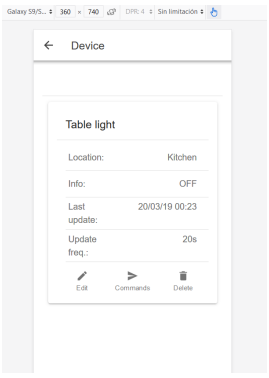
C.3. Pruebas de interfaz



(a) P. I . Login Galaxy S9

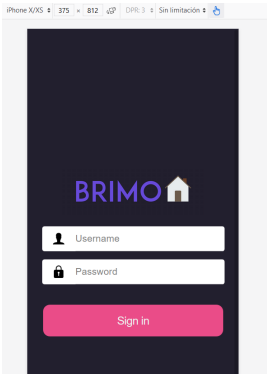


(b) P. I. Listado dispositivos Galaxy S9

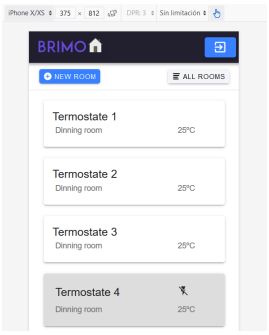


(c) P. I. Vista dispositivos Galaxy S9

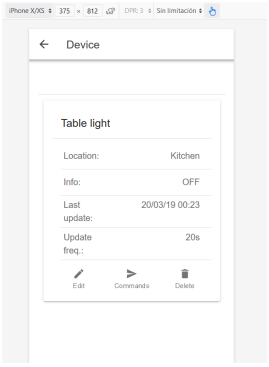
Figura C.10: Pruebas de interfaz Galaxy S9/S9+



(a) P. I . Login iPhone X

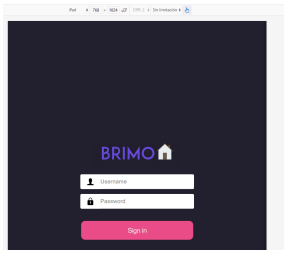


(b) P. I. Listado dispositivos iPhone X

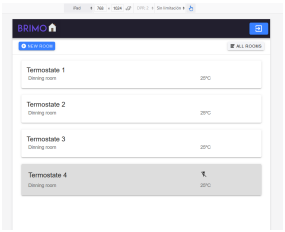


(c) P. I. Vista dispositivos iPhone X

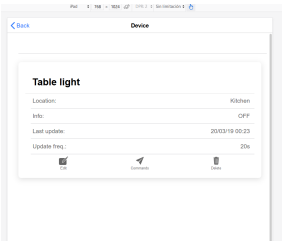
Figura C.11: Pruebas de interfaz iPhone X/XS



(a) P. I . Login iPad



(b) P. I. Listado dispositivos iPad



(c) P. I. Vista dispositivos iPad

Figura C.12: Pruebas de interfaz iPad